In this architecture, the Root Server becomes a communications process server, initiating a single communications process within which it runs the servers for individual services as threads, starting and stopping them in response to client requests and providing process, shared resource and common settings management including fast, low-overhead communication between the dedicated server threads. Each server is run in its own thread and only a single instance of any server is ever running. A number of supporting components implement the messaging abstractions and communications channels which allow passing of messages between running server threads, while the Comms Database provides the shared settings service. (From Symbian OS v9, the CommDB API is provided for compatibility only; the Central Repository should be used for all shared settings.)

The Root Server is responsible for running the following dedicated servers, which implement a common Comms Provider Module (CPM) interface defined by the framework:

- C32 Serial Comms Server
- ETel Telephony Server
- ESock Socket Server
- Resolver Server
- Fax server.

The individual services are described in more detail in the sections that follow. Each service provides a client-side session API, encapsulated in a single static DLL to which clients link. The general usage pattern is thus:

1. Create a client session with the appropriate server, for example the Serial Server or Socket Server; this exposes the server's client-side APIs to the client.
2. Create a client sub-session with an appropriate object, for example a communications port or a socket; this exposes the object APIs to the client.
3. Use the object.
4. Close the sub-session with the object when finished.
5. Close the session with the server when finished.

It is also worth noting that in Symbian OS communications services are provided user-side; in other words, communications services are not built into the kernel. This protects the kernel from resource failures or

badly behaved processes originating from communications services or clients.

Frameworks

As well as implementing server functions, the principal communications servers also provide extensible frameworks, which are at the heart of the communications architecture.

Frameworks provide extensibility at a number of levels, including:

- at the client-interface level (for example, extending core telephony services to enable fax over mobile networks)
- within the protocol stacks at the protocol level (for example, adding the WAP stack or extending core TCP/IP services to enable packet-level security)
- at the network-interface level (for example, adding support for new technologies such as the Bluetooth Personal Area Networking (PAN) profile)
- at the hardware-abstraction-interface level (for example, extending the telephony baseband interface to support CDMA).

All implementations of communications framework plug-ins conform to the Plug-in Framework (i.e. ECom), in other words they are polymorphic DLLs that implement the standard interfaces which enable the Plug-in Framework server to find and load the appropriate modules at run time on behalf of the requesting framework, as well as the communications-specific interfaces required by the specific communications frameworks.

The Comms Services frameworks include:

- C32 Serial Server, which defines CSY virtual serial port modules
- ETel Telephony Server, which defines TSY baseband interface modules
- Socket Server, which defines PRT protocol modules
- Network Interface Manager, which defines AGT interface agent and NIF network interface modules.

In addition, the Comms Framework component defines the CPM interface which is implemented by all of the dedicated communications servers (but not by the Root Server itself).

9.6 Comms Framework

The Comms Framework components implement the infrastructure used by all communications services:

- The Comms Root Server is the primary communications server, responsible for starting and stopping the communications servers

that provide dedicated services and for providing the process context in which all dedicated servers are run.

- The C32 Serial Server and the ESock Socket Server are, respectively, the data communications and socket servers that provide the two direct client interfaces for communications services (all communications services are accessed through sockets and serial communications services are also available directly through the Serial Server).
- The Network Interface Manager and Network Controller are, respectively, the network interface and connection managers that find and set up appropriate network connections requested by Socket Server clients and that are used (indirectly) by all communications services.

The Comms Framework also includes common utility and framework support, including the framework classes that define the Comms Provider Module (CPM) interfaces to which all communications servers conform and specialized messaging and memory management (Comms Channels and MBufs), designed to enable fast inter-thread communications within the communications process including thread-shared memory. (Communications servers run in their own threads inside the single communications process managed by the root server.)

Also included is the Comms Database, which supports the legacy interface used for storing shared communications settings. (New applications should use the Central Repository.) See Figure 9.8.

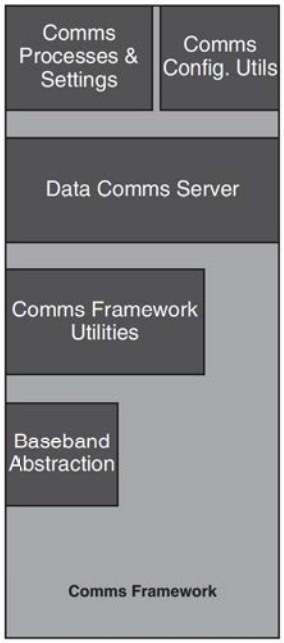


Figure 9.8 Comms Framework components

Design Goals

It is important to remember that on a typical device based on Symbian OS (a mobile phone, for example), all communications must be virtualized over an available, and usually transient, connection. Thus Internet browsing, for example, typically does not take place over a direct Internet connection (as it would on a PC) but is virtualized over telephony or short-link services. As Wi-Fi begins to appear on phones, direct network connections also become possible but very much as complementary options.

The Comms Framework has evolved to provide a generic infrastructure that enables the seamless interoperation of services while providing improved performance, ready for the next generation of high-data-rate services.

Architecture

The Comms Framework sub-block is less a self-contained architectural unit than the architectural glue that binds the different dedicated communications services together. It provides the frameworks that define essential, common communications abstractions, the Root Server that provides the runtime context within which all communications services operate, and the shared settings database and utilities, as well as utilities and libraries, such as the MBuf Manager and Elements components.

The Root Server and Framework Utilities

From Symbian OS v8, all communications servers are implemented as Comms Provider Modules and are run and managed by the Root Server, which loads, configures, runs and monitors CPMs as dedicated threads within the Root Server's own process. Starting the Root Server creates the single communications process and starts the server as the main running thread within in. The Root Server runs from device startup to shutdown.

In Symbian OS, a process is the fundamental unit of protection, with its own address space, while a thread is the fundamental unit of execution, running inside a process and sharing the process address space and any other resources (file handles, for example) with other threads running in the process.

The Comms Framework is the component that provides the abstractions needed to implement Comms Provider Modules including the CPM interface, common thread management and Comms Channels, the asynchronous message queue abstraction that provides an efficient communication mechanism between active CPMs. The CPM framework also defines a file-based configuration method that is used by the Comms Root Server to configure CPMs on loading.

To support implementation of new CPMs, the Comms Elements provides a reusable catalog of common design pattern implementations, for server startup, message passing and generally useful abstractions such as state machines. The MBuf Manager provides a memory management framework that allows direct sharing of data (for example, network packets) between CPMs without copying.

Serial Communications

The C32 Serial Server provides serial services for application and system clients. A key component from the first Symbian OS release, it has been re-architected and re-engineered to support platform security and the new communications infrastructure based on the Root Server. From Symbian OS v8, the C32 Serial Server is a CPM, run and managed by the Root Server. The CPM and supporting mechanisms provide data sharing and efficient inter-server communications without the overhead of running the Serial Server to support other communications services.

The Serial Server follows the standard Symbian OS server pattern, providing serialized access to shared resources. In the simplest case, and unlike other communications servers, clients can gain direct access to the serial hardware on a device by initiating a client session with the server (by making a serial service request to the communications configurator) and then from within the session loading, opening and configuring a (virtualized) serial port. This creates what is, in effect, a raw serial link over the chosen port (either an actual serial port, or virtualized over Bluetooth, infrared, or USB) to another, connected device. Clients can also access serial services through the Socket Server.

As well as providing a client API, the Serial Server defines the framework interface that communications plug-in modules (CSY files) implement. A CSY module is implemented as a polymorphic DLL (with a CSY extension, by convention) that exports a factory function for a `CSerial`-derived `CPort` class object. CSY implementations are supplied for true RS232 serial ports and serial port emulation over IrDA, Bluetooth and USB. At the level below the plug-in modules, logical and physical device drivers implement the hardware-level interfaces.

Sockets

Sockets were first introduced as a networking abstraction in Berkeley Unix (BSD), providing a generic mechanism to associate a communications protocol with a data pipe (dedicated communications channel) connecting two processes, transparently of where the processes were actually running and using a simple, file-type semantics. The ESock Socket Server provides sockets-based communications on Symbian OS through a client session API and an underlying framework for creating and loading protocol implementation plug-ins (PRT files) that determine the type of the

socket and provide the underlying protocol implementations. Sockets-based protocol implementations are supplied allowing services to be run over a wide range of possible bearer protocols including Bluetooth, IrDA, TCP/IP and SMS.

The sockets abstraction provides a common client interface to networking, serial and short-link communications protocols, providing a sockets API plus name and address resolution and connection management.

ESock was originally provided as part of the networking implementation of the first Symbian OS release, but over subsequent releases it has evolved into a more generic mechanism for requesting any communications services. Since Symbian OS v8, the Socket Server presents itself to the Comms Root Server as a collection of CPMs whose purpose is to provide protocol sessions to requesting clients by finding and loading an appropriate protocol module, serving it through a client session to the client, transparently managing the shared data structures and channels used for socket communications, monitoring and cleaning up after thread panics and, generally, performing all necessary housekeeping functions and resource management.

Clients connect to the ESock server with a `Connect()` call and then open a sub-session by calling `Open()` on a socket of the chosen type. The socket type is based on the transport protocol. In response to a socket request from a client, the Socket Server loads an appropriate protocol module (PRT file) that implements the requested protocol.

In Symbian OS v9, the Socket Server is multi-threaded, improving performance.

Network Interfaces

The underlying interface to the network transport layers is provided by the Network Interface Manager, or NIFMan, and its supporting components, which load interface agents (AGT files) to establish network connections and then create an appropriate network interface (NIF file). Connections supported at Symbian OS v9 are either circuit-switched or packet-switched data connections running through telephony services, or an Ethernet implementation running over serial communications or short-link services. The chosen network interface is bound to the TCP/IP stack. NIFMan defines the plug-in framework (i.e. the base classes from which plug-ins must derive and hence the interfaces they must implement) for the network controller modules. A network controller owns both networks and bearers.

The Network Controller component is used by the Network Interface manager to select a suitable outgoing interface, for example from those pre-configured in the Comms Database. It loads first the appropriate agent to establish the physical connection and then the appropriate network interface. Thereafter, data can flow between the requesting client and the

network interface through the loaded PRT stack module and the Socket Server that loads it.

At the lowest level of the networking services are the modules that implement the interfaces to the physical link layer, the plug-ins to the Network Interface Manager (NIF files) and other related low-level plug-ins. Supported interface types include Ethernet, PPP, SLIP and a tunneling NIF, each of which can serve as an interface to different physical link-layer carriers, for example physical cable or infrared implementations of serial communications, Bluetooth, GPRS, and so on.

NIFMan can be thought of as the server that manages the overall control of network and bearer selection, delegating the actual work to the Network Controller and the agents that plug-in to NIFMan. Agents are the workhorses that manage the pairings of networks to bearers. Typical bearers might include:

- a GSM radio network supporting circuit-switched data calls
- a CDMA95 radio network supporting circuit-switched data calls
- a GSM radio network supporting packet-switched data contexts
- a UMTS radio network supporting packet-switched data contexts
- a CDMA2000 radio network supporting packet-switched data contexts
- an Ethernet wired network connection to a LAN
- an 802.11 (Wi-Fi) radio network connection to a LAN.

With multi-homing, there may be multiple access technologies available to reach the same network destination. For example, a given network (an Internet ISP, say) may be reachable by all of the following: circuit-switched data (for example, a GSM data call), packet-switched data (for example, a GPRS connection) or WLAN (directly via Wi-Fi or perhaps via Bluetooth connection to a PC). In contrast, another network (the user's 3G mobile network, for example) may be reachable only by packet-switched data. Multi-homing enables each network and bearer combination to be separately defined, so that the relationship of networks to bearers is no longer 1:1 but one to many (i.e. multiple combinations for a given network, based on all the possible bearers).

Shared Settings

Historically in Symbian OS, the Comms Database, or CommDB, is the repository in which all communications-related settings and configuration information is stored. Settings are used, for example, by system-level components for default host-name resolution and to determine connection preferences, availability of physical modems, services, configured ISPs,

GPRS access points, LAN services, and so on, as well as by applications that, for example, may need to allow users to set or change settings.

As well as containing preferences and settings, CommDB provides the utilities needed to set, store and manipulate settings and to read and write settings into XML formats.

CommDB has been a part of the system since the first Symbian OS releases. In Symbian OS v9, however, its functions are replaced by the Central Repository, to which the CommsDat component provides a communications-specific interface for stored settings. Compatibility is maintained for old-style CommDB requests.

Component Collections

Comms Process and Settings Collection

The Comms Root Server provides the main thread in the communications process and is responsible for starting and managing all other communications process threads. These are started at device boot, rather than on demand, as in previous operating system releases. See Figure 9.9.

Table 9.1 Comms Process and Settings Components

Component Name	Development Name
Comms Root Server	ROOTSERVER

It provides client-side APIs for loading, configuring and binding provider modules; polices any relevant security policies; and publishes a Publish & Subscribe property to notify thread death of provider modules.

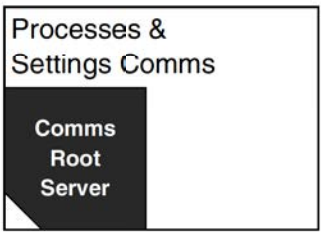


Figure 9.9 Comms Process and Settings components

Comms Configuration Utilities Collection

Communications-related settings and configuration information are used to set and determine the host name, connection and service provider defaults. The Comms Database (CommDB) is the legacy repository that,

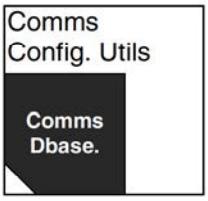


Figure 9.10 Comms Configuration Utilities components

Table 9.2 Comms Process and Settings Components

Component Name	Development Name
Comms Database	`COMMSDAT, COMMDB_SHIM, COMMDB_COMPAT`

from Symbian OS v9, is replaced by the CommsDat interface to the Central Repository, although the CommDB API is preserved for compatibility. See Figure 9.10.

Data Comms Server Collection

This collection contains servers and supporting components that provide the key client interfaces for data communications. See Figure 9.11.

Table 9.3 Data Comms Server Components

Component Name	Development Name
C32 Serial Server	`C32`
ESock Server	`ESOCK`
Network Interface Manager	`NIFMAN, DIALOG`
Network Controller	`NETCON`

- The C32 Serial Server provides the client session APIs and server implementation for serial type communications and the framework

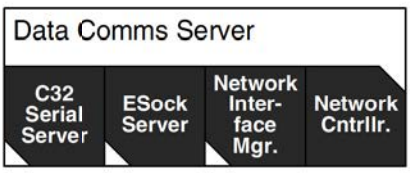


Figure 9.11 Data Comms Server components

for creating and loading the communications plug-in modules (CSY files) that implement the serial-port abstractions, enabling clients to access virtual serial ports independently of the underlying hardware.

- The ESock Socket Server provides the client-session APIs and server implementation for sockets-based communications and the framework for creating and loading protocol implementation plug-ins (PRT files).
- The Network Interface Manager provides the bearer-level support for the Socket Server, providing the framework for creating, loading and managing interface agent (AGT file) and interface plug-ins (NIF files). Interface agents find and load network-interface implementations and bind them to the TCP/IP stack to create the bearer-level connections over which the socket protocols served to clients by the Socket Server actually run.
- The Network Controller is the component that selects a network interface agent to create an appropriate network interface. It reads connection preferences for the client from stored communications settings, based on which it chooses both a network and a bearer (i.e. an access technology). Having made its choice, it loads the appropriate agent. It is implemented as a plug-in library loaded by the Network Interface Manager.

Comms Framework Utilities

These utilities provide framework support for the Root Server and for Comms Provider Module mechanisms. See Figure 9.12.

Table 9.4 Comms Framework Utilities Components

Component Name	Development Name
Comms Framework	COMMSFW
Comms Elements	ELEMENTS
MBuf Manager	MBUFMAN

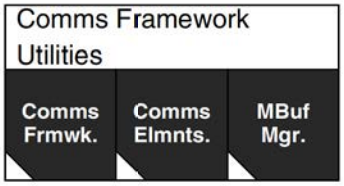


Figure 9.12 Comms Framework Utilities components

- The Comms Framework provides the framework base classes and utilities that support the communications architecture based on the Comms Root Server, including base classes that define Comms Provider Modules (the addressing and binding mechanism used by the Comms Root Server to identify and load modules), the message definitions and communications channel queue abstraction used to communicate between modules and the Comms Root Server, and thread-creation support.
- The Comms Elements are an internal library of ready-made programming patterns, for example state machines and message parsers, that are used within communications services and are made available as reusable objects.
- The MBuf Manager defines and manages MBufs, a communications-specific shared-memory mechanism allowing provider modules (i.e. multiple threads within the primary communications process) to share memory buffers and therefore avoid unnecessary copying of messages and data. For example, MBufs can contain data packets as well as arbitrary C++ objects.

Baseband Abstraction Collection

The Baseband Channel Adaptor (BCA) provides an abstraction of the actual channel used to communicate with the baseband processor, for use by communications components (which, therefore, don't need to understand the actual channel implementation) and a plug-in framework for a hardware-specific interface implementation module. The actual channel is dependent on the hardware design and may comprise a physical fast serial link, USB or other fast bus, a shared memory or even a shared register protocol. See Figure 9.13.

Table 9.5 Baseband Abstraction Components

Component Name	Development Name
Baseband Channel Adaptor	BCA



Figure 9.13 Baseband Abstraction components

9.7 Telephony Services

The telephony architecture was designed to provide flexible support for a wide variety of possible phone types, including conventional analog modems, GSM phones and even desktop phones containing an integrated Symbian OS device.

Like other communications services, the Telephony Services block is organized around a primary server and framework, the ETel Telephony Server, supported by protocol implementations for specific services, low-level plug-in modules implementing hardware adaptation interfaces defined by the framework and some assorted high-level utilities.

The design principle for ETel was to abstract a small core set of universal telephony functionality as the Core API, while providing a flexible extension mechanism to enable support to be added for specific service and network types at both the client interface, enabling support for custom services and at the hardware interface, enabling support for different telephone-baseband implementations. The straightforward goal of the Core API is to enable telephony clients to pass information over a generic phone link.

From this starting point, support has evolved from basic Hayes modem control (AT commands) through GSM 2G standards, to 2.5G (GPRS, EDGE) and CDMA (for the North American market and other markets, such as Korea, that initially adopted CDMA rather than GSM), and to

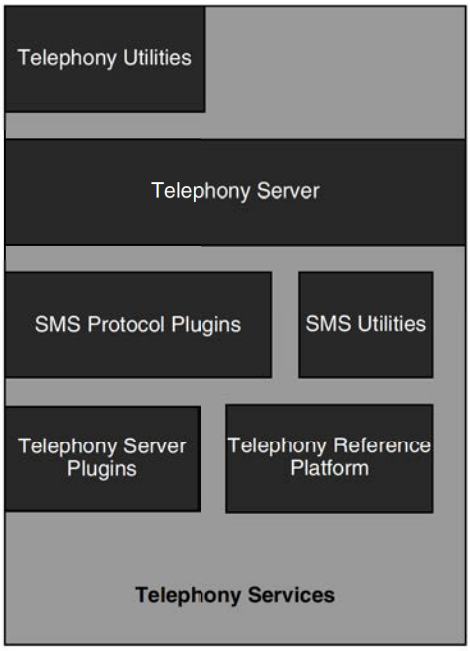


Figure 9.14 Telephony Services components

3G UMTS and CDMA2000 (respectively the 3G evolutions of GSM and CDMA).

From an initial emphasis on dial-up and modem connections, providing a fully integrated telephony service became important as Symbian OS moved onto mobile phones. More recently as mobile telephony has evolved towards packet-based networks, support for high-bandwidth data services has become important. See Figure 9.14.

While the basic phone services in Symbian OS are quite mature and were well established by Symbian OS v6, incremental enhancements have been introduced with almost every release since.

Architecture

Telephony Services are structured around the ETel Telephony Server. ETel provides a core set of common, network-independent telephony services that abstract control of telephony devices either connected to or integrated into a Symbian OS phone and enable client access to phone services. ETel is implemented as an extensible framework into which modules can be added to extend the core functionality at the client level.

The ETel framework also defines the low-level, hardware adaptation interfaces and provides the mechanisms that support hardware adaptation plug-in implementation modules (TSY files).

Conceptually, the ETel core API is extensible in two directions: in the direction of hardware, supporting new networks, baseband implementations and other hardware evolutions, and, in the client API direction, enabling new services to be supported.

While extensibility implies flexibility, it also implies a significant division of labor between Symbian and licensees to extend the telephony support appropriately for a given phone or family of phones:

- On the Symbian side, the ETel server core and Multimode framework support extensions for new standards (which is how the initial GSM-only support has been extended first through CDMA and then to 3G UMTS and CDMA2000) and expose the TSY provider module plug-in API.
- On the licensee side, the Telephony Application and low-level TSY provider modules support platform- and device-specific customization.

Licensees implement a custom TSY and any additional custom APIs they choose to add to support unique features of their own telephony hardware.

In addition, licensees provide engine support and custom UIs, for example, for phone security (such as PIN-based locking of the phone application), and the phone application itself, which must include a

platform-specific user interface and must also support comprehensive user-interface-independent functions including handling networks, audio, contacts, logging and call handling, number parsing, and so on. Telephony Services includes a number of libraries and utilities that provide basic support for such functions.

Note that the 'licensee' may be either a platform vendor such as S60 or UIQ, providing a pre-integrated user interface and application suite solution to its customers, a phone vendor (or consortium such as FOMA) creating a bespoke UI and applications, a third-party developer of a phone application, or a hardware partner providing a packaged phone hardware solution.

Evolution of Mobile Services

Mobile phone services and technologies have evolved rapidly, as has the global market for mobile phones, including significant cycles of boom and bust. Basic mobile network technologies have evolved from 'plain old' GSM through GSM Phase 2+, otherwise known as 2.5G (GSM, GPRS, EDGE), to UMTS 3G, with similar evolutions from CDMA to 3G CDMA2000. Symbian OS has tracked these evolutions. It enables control of landline and mobile phone modems and supports wireless telephony standards for all markets.

GSM uses a packetized but synchronous Time-Division Multiple Access (TDMA) approach to sharing available bandwidth between multiple users. Voice is digitally encoded and transmitted as digital packets in timeslots (frames) at a data rate approximating 19 200 baud (equivalent to modem speeds from around the late 1980s).

Support for basic GSM services requires support for receiving and making voice calls, receiving and sending SMS messages, showing that SMS messages have been received, and receiving and making circuit-switched data calls, for example fax calls. GPRS adds the requirement to support making and receiving packet-switched data calls.

EDGE and 3G networks extend these requirements to include, for example, both one-way and two-way audio and video calls including support for two-way tele-conferencing; streaming of audio and video to a phone; interactive, session-like two-way request–response (for web browsing or remote database query); and background data delivery for example of SMS messages.

GPRS and EDGE add packet data services by stitching together multiple GSM voice channels to create a higher bandwidth channel. GPRS provides data rates up to 170 kbps, which EDGE improves by a factor of three (either in speed or in the number of simultaneous subscribers supported at GPRS data rates).

Both GSM and CDMA remain circuit-oriented, voice-centric technologies. UMTS evolves GSM to use Wideband CDMA to gain higher

data rates. Unlike GSM or CDMA, UMTS is fully packet-switched, not circuit-switched.

Historically, CDMA has dominated the North American market, while GSM originated as a European standard that has had widespread global uptake. GSM has also recently increased its market share in many CDMA-dominated markets to become a second-line network technology.

Telephony Server

The ETel Telephony Server manages access to telephony functions on a Symbian OS device, regardless of the details of the available phone hardware. Indeed, there may be no onboard phone hardware, as was the case in the first Symbian OS devices. As well as supporting fully featured mobile phones, ETel supports the use of data ports thus enabling two-box solutions, for example using a mobile phone as a modem via infrared or Bluetooth, which was an early use case.

The server implements the standard Symbian OS client–server framework, providing a client-side API (as a separate DLL to which clients link). The server also implements the CPM interface and is thus a communications provider that is managed and run by the Comms Root Server and which provides thread management and communications channels for fast communication with other communications server threads.

The basic abstractions made available by the Telephony Server are phones, lines and calls. The server also provides an extension framework, which is used to add extended client services and a low-level hardware adaptation interface that is implemented by hardware adaptation plug-in modules. Clients open a server session with the Telephony Server and then open sub-sessions with phone, line and call objects.

The Core API includes generic functions for requesting the capabilities or status of the phone hardware and making and managing voice, data and fax calls.

Basic telephony extensions supporting GSM/GPRS are implemented by the ETel Multimode extension and other extension modules supply further CDMA, messaging, 3GPP packet data and fax-specific extensions. Collaborating components are all realized inside sub-sessions or the root ETel server session to a client, that is created when the ETel server is started by the Comms Root Server in response to a client request.

The ETel Server and Core API, together with the Fax Client–Server, formed the basis of the original telephony implementation in ER5. The ETel Core API was rearchitected in Symbian OS v7 when the other extensions were introduced and most were further enhanced in Symbian OS v8.

ETel Third-Party API

The ETel third-party API was introduced in Symbian OS v7 to provide a restricted but common 'safe subset' of telephony functionality to

third-party (i.e. non-licensee and partner) application developers. It was significantly extended in Symbian OS v8, adding support for multiple voice calls, better access to onboard and network status information and system notifications and events, and access to IMEI and IMSI numbers.

Telephony Messaging

The ETel Multimode extension includes generic support for telephony messaging, with specific implementations (for example for GSM, CDMA and WAP) implemented as Socket Server protocol-module plug-ins, providing a common sockets-based interface to messaging clients. The protocol modules perform the actual encoding and decoding of messages, support SIM card message store management functions and interact with the Telephony Server (via ETel Multimode) for transmission and reception of message.

SMS messaging clients include the messaging application support components at the application-services level, for example the SMS MTM, CDMA MTM and Java messaging components.

Similarly, the WAP Stack is a client for WAP messaging, typically to expose a Wireless Datagram Protocol (WDP) service to a WAP client. The WAP protocol module in turn directly cooperates with the SMS protocol module, which undertakes the transaction with the Telephony Server.

Note that the Telephony Server may not be the ultimate provider of the message service, for example if an SMS is requested to be sent over a Bluetooth link. In this case, the Telephony Server creates a further Socket Server session requesting the appropriate bearer and the messaging interface is a serial port plug-in for the appropriate bearer rather than the TSY interface to onboard phone hardware.

Part of the messaging support consists of utility classes that implement encoding and decoding functions and streaming, logging and backup-server interface classes. Utilities are provided as standalone DLLS (linked to by clients at compile time).

Interfacing to the Baseband

TSY modules are the telephony equivalent of the Serial Server's CSY virtual serial-port implementation modules and are defined and loaded by the ETel Framework. A TSY is an ECom (plug-in framework) compliant plug-in that provides the glue between Symbian OS Telephony Services and the phone baseband (the telephony stack).

The Telephony Server passes client requests made to it on sub-session objects (which may be based on the Core API, Symbian-supplied extension APIs, or custom APIs created by licensees by extending the core framework) to the TSY, which translates them into proprietary requests the baseband understands. The TSY plug-in model is a direct borrowing of the CSY model used by the Serial Server.

The Telephony Server framework provides the abstract base classes for each of the objects implemented by a TSY, representing phones, lines, calls, faxes and extensions.

Symbian OS supplies four TSYs as reference implementations. The Multimode TSY shipped for the first time in Symbian OS v7, as an upgraded and renamed version of the original GSM.TSY that shipped with ER5, incorporating GPRS support. The CDMA and SIM TSYs also shipped for the first time in Symbian OS v7, as did a first version of a Telephony Reference Platform (TRP) TSY. The Symbian OS v9 TRP TSY runs on Texas Instruments H2 development board hardware but is designed to be easily ported to other platforms.

Component Collections

Telephony Utilities Collection

This collection contains helper components that use the telephony server but which are not used by it. See Figure 9.15.

Table 9.6 Telephony Utilities Components

Component Name	Development Name
Telephony Watchers	`TELEPHONY_WATCHERS`
Phonebook Sync	`PHBKSYNC`
Dial	`DIAL`

- The Telephony Watchers are watcher Framework plug-ins that monitor telephony conditions and report them as Publish and Subscribe properties, including current signal strength, battery level and whether a call is in progress. They were introduced in Symbian OS v8.
- The Phonebook Sync server enables synchronization of contacts between a phonebook application and entries stored in the Integrated Circuit Card (ICC) or 'SIM' card of a device. It was originally introduced as part of the ETel Multimode extension 3G support for UMTS and CDMA2000.

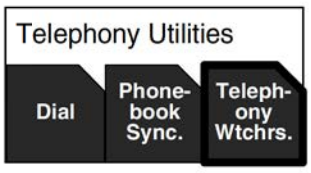


Figure 9.15 Telephony Utilities components

- The Dial component consists of dialing utilities the use of which is deprecated from Symbian OS v9.

Telephony Server Collection

The ETel telephony server and core framework implements the basic telephony functions and is extended by the Multimode framework into a uniform generic API for all mobile telephony independent of the underlying network, with additional packet-data extensions for 2.5G and 3G packet services, CDMA-specific extensions, plus SIM Toolkit utilities, fax support and a third-party API that opens a common subset of telephony functions to third-party application developers. See Figure 9.16.

Table 9.7 Telephony Server Components

Component Name	Development Name
ETel Server and Core	`ETEL`
ETel 3rd Party API	`ETEL3RDPARTY`
Fax Client and Server	`FAX`
ETel Multimode	`ETELMM`
ETel Packet Data	`ETELPCKT`
ETel SIM Toolkit	`ETELSAT`
ETel CDMA	`ETELCDMA`

- The ETel Server and Core API provides clients with access to telephony functions on Symbian OS. It implements the standard Symbian OS client–server framework, providing a client-side API (as a separate DLL to which clients link). The server in turn translates these into TSY requests, which are passed on to a TSY module. The server dynamically loads and unloads TSY modules at client request. The TSY implements a customized interface to the onboard hardware (although in a two-box case, it would route back through an appropriate socket to

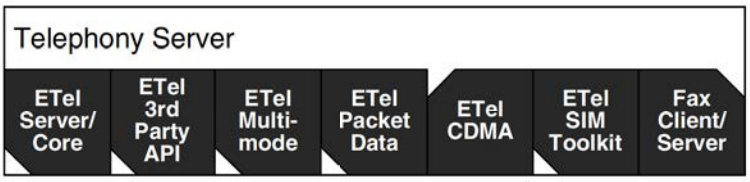


Figure 9.16 Telephony Server components

use the requested communications port). Like other communications servers, ETel is a CPM that runs as a thread within the communications process.

- The ETel Multimode component extends the Core ETel API to provide, as far as possible, a uniform API, for making voice, fax, data or multimedia calls, that is independent of the underlying mobile network and phone architecture (e.g. 2G, 2.5G, or 3G).
- The ETel 3rd Party API is implemented as a sub-session providing a subset only of the Core ETel, ETel Multimode and ETel Packet APIs. Unlike the other main telephony APIs, which are restricted to licensees, the ETel 3rd Party API is open to third-party developers, enabling them to make applications that can use the telephony features or create dedicated, phone-aware applications.
- The ETel CDMA component extends the ETel multimode sub-session to implement a high-level API for CDMA-specific telephony applications.
- The ETel Packet Data component is an ETel Telephony Server extension framework enabling access to GPRS Release 97/98, CDMA/CDMA2000, Release 99 (GPRS and UMTS) and Release 4 (UMTS) packet services. It enables clients to configure, modify and activate a PDP context for a network packet-switched service and to control a packet-switched connection.
- The ETel SIM Toolkit provides the functionality of a GSM/WCDMA (U)SIM Application Toolkit, implemented as a sub-session.
- The Fax Client and Server is not an ETel extension. The server is a DLL that provides a framework for adding fax functionality to applications and is driven by the Fax Client via ETel and a suitable TSY. The Fax Client is accessed by applications through the Messaging Send-As API.

SMS Protocol Plug-ins Collection

These protocol modules and other plug-ins implement telephony-based messaging for GSM and CDMA SMS and WAP messaging. See Figure 9.17.

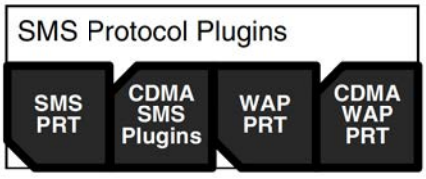


Figure 9.17 SMS Protocol Plug-ins components

Table 9.8 SMS Protocol Plug-ins Components

Component Name	Development Name
SMS PRT	`SMSSTACK`
WAP PRT	`No unit`
CDMA SMS Plug-ins	`CDMASMSSTACK`
CDMA WAP PRT	`No unit`

- The GSM SMS PRT protocol module enables its clients to send and receive GSM SMS messages, enumerate and delete messages from phone stores and read and write SMS parameters on the SIM. It is implemented as an ESock plug-in protocol module, therefore clients interact with it though an instance of `RSocket`; all operations are initiated by IOCTL calls on `RSocket`.
- The CDMA SMS protocol implementation conforms to IS-637 and supports Wireless Paging, Wireless Messaging, Voice Mail Notification, Broadcast SMS, Service Category Programming, Wireless Enhanced Messaging and Card Application Toolkit Protocol.
- The WAP PRT protocol module is used by the WAP Stack for sending and receiving SMS messages.
- The CDMA WAP PRT provides functional equivalents of the GSM WAP protocol module.

SMS Utilities Collection

The GSM Utilities and SMS Utilities components are used by the SMS protocol modules (the SMS stack) to assist in creating and processing SMS messages. For example, GSM Utilities includes encoding and decoding routines and SMS Utilities includes streaming classes (to stream message objects across the Socket Server), logging classes and interfaces to the backup server. They are implemented as utility DLLs that are linked to by clients. See Figure 9.18.

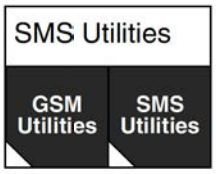


Figure 9.18 SMS Protocol Utilities

Table 9.9 SMS Protocol Utilities

Component Name	Development Name
GSM Utilities	GSMU
SMS Utilities	SMSU

Telephony Server Plug-ins Collection

This collection contains reference telephony server plug-in modules (TSY files), loaded by the Telephony Server. See Figure 9.19.

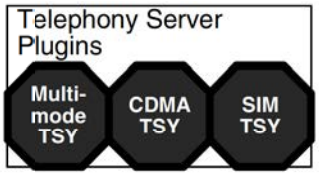


Figure 9.19 Telephony Server Plug-ins

Table 9.10 Telephony Server Plug-ins

Component Name	Development Name
MultiMode TSY	`MMTSY`
CDMA TSY	`CDMATSY`
SIM TSY	`SIMTSY`

- The MultiMode TSY provides the GSM and GPRS functionality. It uses the AT command interface to communicate with the phone or modem via standard AT commands over a serial or infrared link.
- The CDMA TSY is the CDMA equivalent of the MultiMode TSY for GSM. The ETel TSY reference plug-in for CDMA is replaced on an actual device by a hardware-specific licensee TSY.
- The SIM TSY is a simulator module designed to enable automated testing of a range of operating-system components in a simulated GSM, CDMA and WCDMA mode. It does not communicate with any real hardware (neither a modem nor a phone) but instead uses static configuration data and dynamic system-agent notifications to simulate the presence of phone hardware. It supports the Core ETel API, Multimode ETel API and ETel Packet API requests.

Telephony Reference Platform Collection

These components support a standard reference platform telephony implementation. See Figure 9.20.

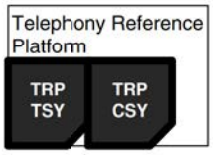


Figure 9.20 Telephony Reference Platform components

Table 9.11 Telephony Reference Platform Components

Component Name	Development Name
TRP TSY	`TRP`
TRP CSY	`TRP`
Baseband Channel Adaptor for C32	`C32BCA`

- The TRP TSY is a reference TSY designed to run on development-board hardware, as part of a wider effort to make easier licensee development on phones using Symbian OS.
- The TRP CSY is used to manage the internal channel between the telephony hardware (a dedicated phone-side core running the TI phone stack) and the application hardware (an ARM core running Symbian OS). The logical driver on the TI H2 board presents the internal serial bus as a standard serial port.
- The Baseband Channel Adaptor for C32 is a reference plug-in providing a serial communications implementation of the Baseband Channel Adapter interface (see Section 9.6). For example, the Telephony Reference Platform provides a serial communications BCA plug-in implementing the BCA interface.

9.8 Networking Services

Web browsing and email were the functions that motivated the inclusion of networking services in the first releases of Symbian OS, although the potential for more exotic applications such as network news readers and multiplayer games was.

It is worth remembering in this context that the devices at which the first ER5 release was targeted were not phones, although they were connected and they were telephony-enabled in the sense that they were designed to interoperate with phones. That interoperation, however, was understood in terms of phone-as-modem, dialing up an ISP to access an email account, a corporate intranet or the Internet, or even the web. Neither email nor the Internet were ubiquitous in the way that they both now are and the web was still very much a novelty.

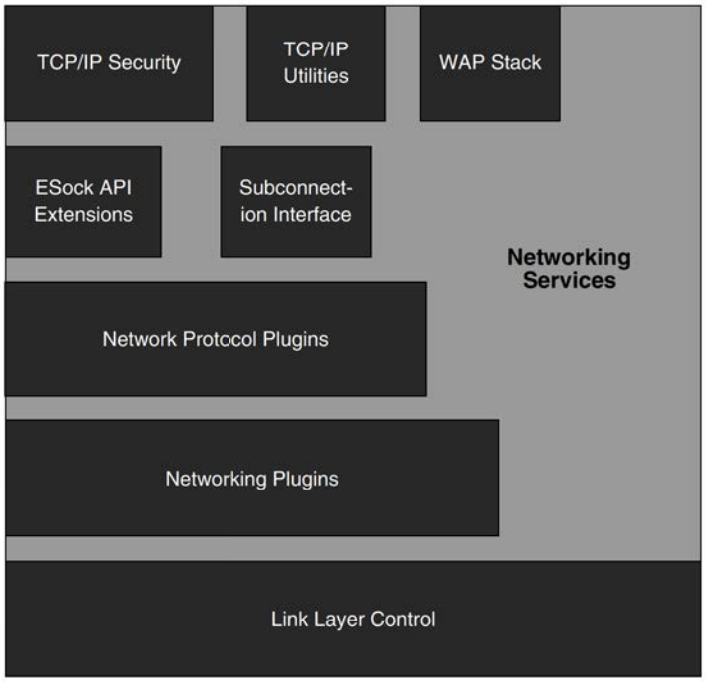


Figure 9.21 Networking Services components

The core of the networking implementation remains the TCP/IP v4/v6 networking stack, implemented as a PRT Socket Server plug-in module and the network interface plug-ins that support it and which are, in turn, supported by link-layer plug-ins.

While the ESock Socket Server and Network Interface Manager have migrated out into the Comms Framework to provide generic socket support for all communications services (and not just for networking), networking services have expanded to encompass TCP/IP enhancements such as IPSec, telephony-driven networking enhancements including packet-data services (for GPRS and UMTS) and Quality of Service (QoS, required for 3G services), as well as completely new technologies such as WAP and most recently (in Symbian OS v9) Wi-Fi. See Figure 9.21.

Networking Stack

Symbian OS Networking Services are based on a TCP/IP protocol implementation, TCP/IPv4/v6 PRT (in effect the transport and network layers of the OSI seven-layer model), together with IP extensions that implement various packet-level services including QoS and IPSec. See Figure 9.22.

However, TCP/IP packets and the stack itself are not directly available. TCP/IP packets are encapsulated within the stack and there are no visible TCP/IP packet classes, for example. The stack is implemented as a Socket

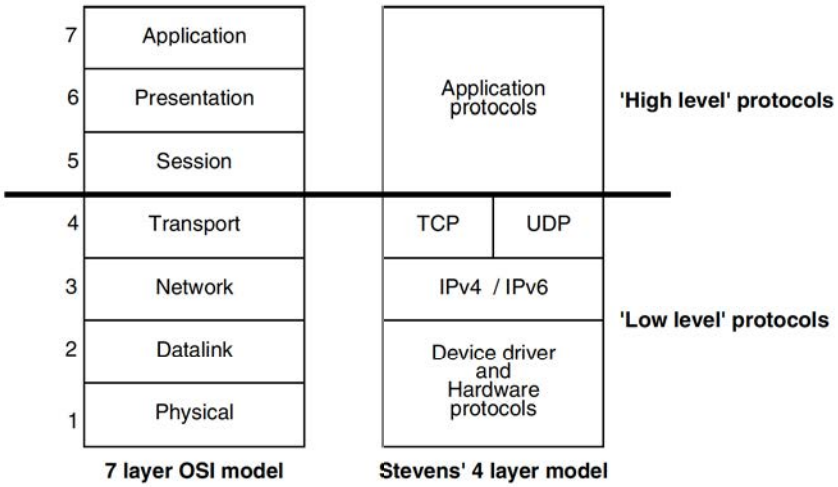


Figure 9.22 The OSI Seven-Layer model and simplified layer model

Server PRT protocol plug-in, and network services are made available through the sockets interface, by requesting a TCP/IP socket.

The stack does however support a hook mechanism provided by IP Hook to enable packets to be accessed within the stack on the inbound and outbound paths, for example to allow pre- and post-processing and other packet transformations; that, for example, is the mechanism used to implement IPSec packet-level encryption and decryption.

A socket is a session-based abstraction that sits logically above the networking protocol implementation, which provides the transport and network layers implementation.

The bottom interface of the stack relies on the Network Interface Manager to select a suitable outgoing interface, which in turn relies on the Network Controller to find a network agent to negotiate the chosen connection (for Network Interface Manager and the Network Controller see Section 9.6).

A number of network agents (AGT files) are available: CSD.AGT, to establish circuit-switched data connections; PSD.AGT, to establish packet-switched data connections; and NULL.AGT, which implements a minimal agent that is used with Ethernet.

A number of network interface implementations (NIF files) are available, including for PPP and Ethernet, as well as a QoS Test NIF that is used in conjunction with the QoS Framework.

At these levels, the architecture has evolved quite significantly since Symbian OS v7, which implemented a rather simplistic view of networks and connections. Particularly with UMTS packet-switched 3G networks, the networking world becomes more complex. For example, multi-homing means that devices can have multiple IP addresses (multiple

network interfaces may be active, each with its own IP address and potentially each on a different network) and packet-switched phone data services mean that multiple interfaces and networks may provide access to a single network destination. However, from an application perspective these changes are mostly invisible and impact only systems developers.

Note that Bluetooth and wireless LAN are not supported by default but require comparable drivers to abstract the hardware for the (overlying) Ethernet NIF implementation. The NIF can support one lower-layer packet driver, loaded during initialization.

PPP NIF has been part of the networking delivery since the first releases of Symbian OS but was significantly enhanced in Symbian OS v8 to improve interoperability with MS Windows, for example by supporting Microsoft extensions to CHAPS dialup authentication.

The Tunneling NIF was introduced with VLAN support and IPSec reimplementation in Symbian OS v8.

Network Security

Network Security protocols operate at different levels in the overall networking stack. TLS and SSL (the two can be considered synonyms) operate at the transport level, providing per-packet encryption and decryption. IPSec on the other hand operates at the network level and is principally designed to support secure networks, for example Virtual Private Networks (VPN) based on policy.

The TLS component implements TLS v1.0 (Transport Level Security) and SSL v3.0 (Secure Sockets Layer), providing more or less transparent, per-packet encryption-based security to client applications, for example HTTPS or SyncML. TLS is implemented as a number of separate DLLs exposing client APIs to applications, which enable sockets to be secured and internal APIs used by networking and security components. TLS was first introduced in Symbian OS v7 and was redesigned and enhanced in Symbian OS v8. A typical use of SSL is to enable secure browser-based transactions.

The IPSec implementation provides security policy management, including support for multi-homed clients (so that different security policies can be associated with the different IP addresses in use by the device) and multiple active policies. IPSec is implemented as a policy server and supporting libraries, as well as a protocol-level PRT Core IPSec PRT plug-in. In effect, it sits between the Socket Server and clients requesting secure sockets. The IPSec PRT does not implement the full interface required by the Symbian OS v9 Socket Server architecture based on the Comms Framework and therefore is considered to be a 'pseudo-PRT'.

IPSec uses the networking stack Hook interface to inspect all incoming and outgoing packets and apply the required cryptographic transformations. The actual security algorithms and libraries are implemented by cryptography and security services components.

The VPN component uses IPSec to manage VPN policies and connections, including VPN password management and is implemented as a VPN manager server and supporting libraries.

IPSec was first introduced into Symbian OS in v7 and was redesigned and enhanced in Symbian OS v8.

Quality of Service

Quality of Service enables performance characteristics to be specified for a communications channel in a packet-based network, to ensure that the required data rates for a given application are met.

The QoS Framework PRT implements QoS policy setting for open sockets (Socket Server sub-sessions) that are treated as QoS channels, providing generic and UMTS-specific APIs for use by applications. While GPRS supports general QoS principles, UMTS defines four traffic classes (Conversational, Streaming, Interactive and Background). Like IPSec, QoS is considered to be a 'pseudo-PRT'.

Networking Daemons

A number of standard networking daemons are implemented as part of networking support.

- DND (i.e. DNS) and DHCP are implementations of the Internet standard protocols for domain-name resolution and dynamic host-address assignment.
- DND makes DNS queries to the (remote) network and listens for and responds to local queries. Like its Unix counterpart, it is implemented as a server 'daemon'. It is accessed through the `RHostResolver` class by Socket Server clients (i.e. as a Socket Server sub-session) and supports `GetByName` and `GetByAddress` queries.
- DHCP enables a device to obtain an IP address and network parameters dynamically from the network, so that having a fixed IP address becomes unnecessary. The DHCP implementation consists of a server daemon and a client-side interface and provides a limited API sufficient for the Network Interface Manager to configure an appropriate network interface. (It is not intended for other users.)

WAP Support

Wireless Application Protocol (WAP) evolved out of work started by the Unwired Planet consortium, which evolved into the WAP Forum in 1998 at around the time that Symbian joined it and into the Open Mobile Alliance (OMA) in 2002. WAP was a deliberate attempt to create a Web

standard targeted at mobile devices in general and phones in particular, to make web content browsable on those devices.

WAP defines a protocol stack, much like TCP/IP, with transport and datagram layers defined over a variety of possible mobile phone network bearers. At the top of the stack, the wireless session protocol (WSP) behaves like a binary-encoded HTTP. Unlike HTTP, WAP enables both pull and push models. In the pull model, clients make requests to a WAP gateway that responds by sending data. In the push model, the gateway pushes data to the client, without a client request.

While the full WAP stack consists of multiple protocols, WAP Datagram Protocol (WDP) is the critical underlying mechanism, defining a binary encoding for datagrams over a bearer network, which can be any of GSM, CDMA, SMS, GPRS or 3G network protocols.

Symbian supplied a full WAP-stack implementation in Symbian OS v7. However, where licensees supplied a WAP browser it was generally tightly coupled to a particular WAP-stack implementation and where they didn't the Symbian stack was redundant in any case. Therefore from Symbian OS v8, Symbian OS implements a 'short' stack that only supports WAP messaging features (which, for example, are used by the Multimedia Messaging Service), providing connectionless WAP Push, connectionless WSP, and WDP.

The implementation consists of the Messaging API and the WAP Short Stack, which is supplied as a reference implementation only. These provide the client APIs and implementation for WAP messaging over GSM SMS bearers. (Note that the mapping of WDP to CDMA SMS not implemented.)

An important use case for WAP is as the carrier for MMS delivery. Unlike SMS, which is transmitted over network-signaling channels, MMS uses data-traffic channels and, hence, requires a transport technology (SMS relies on network-specific signaling mechanisms as its bearer). An advantage of WAP is that it provides a uniform transport protocol regardless of the underlying network type (GSM, GPRS, CDMA or 3G).

WAP push is based on notification sent to the terminal over SMS, followed by a WSP 'get' call to fetch the message. MMS is therefore independent of the network type (because WAP implementations run on all network types) and interoperable (because TCP/IP is used on the network side to link WAP gateways and can link gateways on different networks, for example, GSM and CDMA or UMTS).

Architecture

The general structure of Networking Services in Symbian OS will be recognizable to those familiar with the standard OSI 7 layer networking model and corresponds roughly to a utility layer plus the lower four layers. See Figure 9.23.

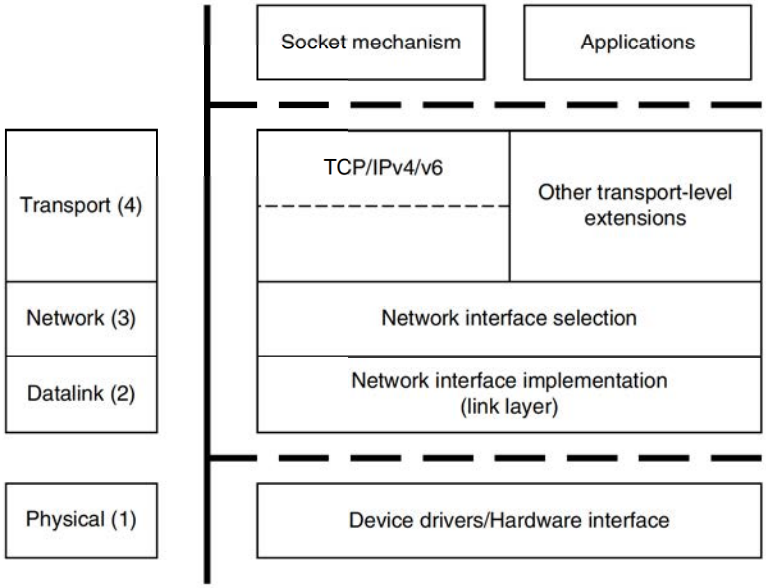


Figure 9.23 Networking Services mapped against the OSI model

The higher layers of the OSI model are mapped by components in the higher layers of Symbian OS, particularly in the Application Services layer.

The OSI model is a generic abstraction and not a rigid specification but understanding the mapping helps to understand the Symbian networking implementation. Roughly, the Symbian OS layers are as follows, working top down:

- networking services and utilities including network security, network daemons, plus the WAP stack implementation
- network-specific extensions to the Socket Server and Network Connection Manager
- core network protocols, the TCP/IP stack and its extensions
- network interface management agents
- network interface implementations.

The higher-level components provide application-level interfaces, the middle-layer protocol implementations are tightly bound to the sockets abstraction, through which all networking services are accessed, and the lower levels provide the interfaces to the available communications bearer technologies.

From a client perspective, the complex interactions between the networking components, the communications framework and the short-link and telephony bearer services are hidden behind the Socket Server. However, it is useful to have at least a general picture of the pattern of interaction.

When a Socket Server sub-session is opened by a client requesting a TCP/IP protocol socket, the request is passed to the TCP/IP stack, which tries to start an outgoing connection. If the stack fails to find an interface that will allow it to reach the selected destination, it reports its failure back to the Socket Server, which then requests the Network Interface Manager to load and start a connection agent of suitable type. Depending on its type, the agent requests a connection from either the Telephony Server or the C32 Serial Server. When the connection is established, the Network Interface Manager loads and starts a NIF module, which implements the required Network Interface and negotiates authentication and other link characteristics (for example, encapsulation and compression) and finally acquires an IP address. The Network Interface manager then binds the NIF to the TCP/IP stack.

Design Goals

The original design goals of Symbian OS Networking Services were based on dial-up access to a network via either a fixed-line modem or a mobile phone. The expected networking applications were standard Internet and web applications, for example email and browsing. Adequate data throughput and the ability to virtualize networking services over an available serial bearer were the key considerations.

Network protocols were also considered important as a way of standardizing support for connectivity with desktop computers for data synchronization and backup.

The increasing specialization of Symbian OS for mobile phones, and the evolution of mobile phones into true network devices as packet services have begun to dominate, has required almost continuous evolution in the architecture and implementation of Networking Services, to keep in step with rapid technological advance, rapid adoption of advanced technologies into mobile phones and the push to provide infrastructure for new services and applications.

Networking has become mainstream for telephony as the basis for high data throughput services such as two-way video conferencing and audio and video streaming. Direct network connection over Wi-Fi is also rapidly becoming a support requirement for mobile phones.

VoIP pushes these trends to their logical step, in effect subsuming telephony into networking.

Component Collections

TCP/IP Security Collection

These components implement secure networking, supporting transport-level security (security at the level of individual IP packets) and connection-oriented security (which is used, for example, to provide VPN support services to application clients running on Symbian phones). See Figure 9.24.

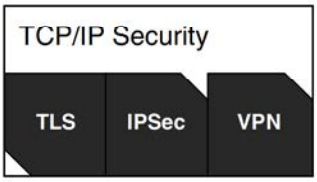


Figure 9.24 TCP/IP Security components

Table 9.12 TCP/IP Security Components

Component Name	Development Name
TLS	`TLS, TLSPROVIDER`
IPSec	`IPSEC`
VPN	`VPNAPI, VPNCONNAGT, VPNMANAGER`

- The TLS component is an implementation of Transport Level Security (TLS) including SSL Secure Sockets, which provide encryption per packet, supporting application-level encryption and authentication-based security, for example for secure web services. Client authentication is based on key management and certificate handling, including support for external cryptography modules ('secure tokens'), for example based on a phone smart card.
- The IPSec component operates at a lower level (i.e. network level) and is principally designed to enable secure networks, for example VPN, based on policy.
- The VPN component provides policy-based connection management and gateway interoperability for VPN connections, i.e. it enables users to connect to VPNs.

TCP/IP Utilities Collection

This collection contains implementations of standard networking 'daemon' server utilities. See Figure 9.25.

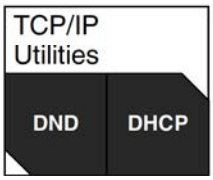


Figure 9.25 TCP/IP Utilities

Table 9.13 TCP/IP Utilities

Component Name	Development Name
DND	DND
DHCP	DHCP

- DND is a DNS implementation that makes DNS queries to the network and listens for and responds to local queries.
- The DHCP component is a Dynamic Host Configuration Protocol (DHCP) implementation used by the PAN Profile and other networking components.

WAP Stack Collection

Symbian provided a full WAP stack implementation in Symbian OS v7. Versions later than Symbian OS v8 implement only a 'short' stack, providing client APIs for connectionless WSP, connectionless Push and WDP. See Figure 9.26.

Table 9.14 WAP Stack Components

Component Name	Development Name
WAP Message API	WAPMESSAGE
WAP Short Stack	WAPSTACK

- The WAP Short Stack component is a cut-down WAP stack supporting WAP messaging, that is, WAP datagrams, WAP Push messaging and

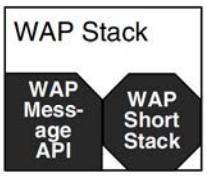


Figure 9.26 WAP Stack components

WSP but not full WAP browsing. It is supplied only as a reference implementation. Vendors replace it with their own short or full WAP stack implementations.

- The WAP Message API implementation provides APIs for WAP Push, connectionless WSP and WDP datagrams.

Sockets API Extensions Collection

The Internet Sockets component is a DLL that provides a library of utility classes and generally useful constants, which specifically support using Internet sockets, to store and manipulate IP addresses, routes, and so on.

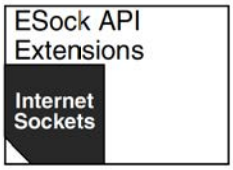


Figure 9.27 Sockets API Extensions

Table 9.15 Sockets API Extensions Components

Component Name	Development Name
Internet Sockets	`INSOCK`

Clients access the Internet Sockets through the generic sockets client API and use the TCP/IP-specific utility classes to perform the IP-specific manipulations. Clients link against the Internet Sockets library. See Figure 9.27.

Subconnection Interface Collection

This is a utility component used by QoS clients to create and package the QoS parameter list. Parameters are set using the `RQoSChannel` class. See Figure 9.28.

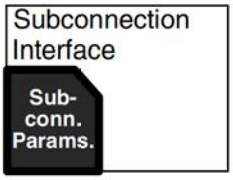


Figure 9.28 Subconnection Interface

Table 9.16 Subconnection Interface Components

Component Name	Development Name
Subconnection Parameters	

Network Protocol Plug-ins Collection

This collection contains the core TCP/IP functionality including the TCP/IP stack, which supports both v4 and v6 standards, the hook mechanism that allows access to packets for inline processing (for example allowing packets to be encrypted 'in place' as they flow through the stack), and IPSec and QoS implementations. See Figure 9.29.

Table 9.17 Network Protocol Plug-ins

Component Name	Development Name
IP Event Notifier	`IPEVENTNOTIFIER`
TCP/IPv4/v6 PRT	`TCPIP6`
IP Hook	`INHOOK6`
QoS Framework PRT	`QOS, QOSLIB, PFQOSLIB, SBLPAPI`
Core IPSec PRT	`No unit`

- The IP Event Notifier PRT is implemented as an IP Hook and raises events to clients based on state changes in the TCP/IP stack. It is principally used by DHCP to determine when and how to perform address negotiation.
- The TCP/IPv4/v6 PRT supplies the core protocol implementations for TCP/IP networking including the IPv4 and IPv6 stacks, TCP, UDP, ICMP and ARP protocols, a Hook interface allowing access to packets, IPSec and QOS protocol modules, and an event notifier service.
- The IP Hook PRT defines an interface to which modules bind to perform transformations on inbound and outbound packets, respectively

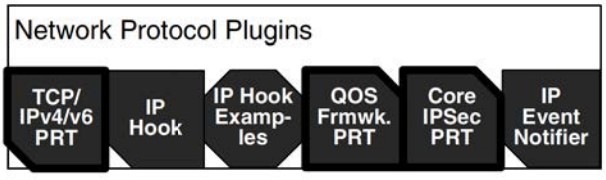


Figure 9.29 Network Protocol Plug-ins

upon receipt from or before delivery to the Network Interface. IPSec is such a Hook, inspecting all incoming and outgoing packets and applying cryptographic transformations as specified in the Security association database.

- The QoS Framework PRT is a Hook module, implementing QoS channels through which it schedules packets. Additional plug-ins map the desired QoS characteristics to relevant link technology.
- The Core IPSec PRT implements core functionality for IPSec in a multi-homed context, that is multiple active network interfaces, for simultaneous use by multiple applications, providing tunnel modes and various high-level APIs. It includes a cryptographic library module, policy managers and parsers.

Networking Plug-ins Collection

This collection contains the network interface agents (AGT files). Two additional components are also included, the Bluetooth PAN profile and the GPRS/UMTS QOS PRT (which is considered a 'pseudo PRT'). See Figure 9.30.

Table 9.18 Networking Plug-ins

Component Name	Development Name
Connection Provider Plug-in	IPCPR
CSD AGT	CSDAGT
PSD AGT	PSDAGT
NULL AGT	NULLAGT
GPRS/UMTS QOS PRT	GUQOS
Bluetooth PAN Profile	BLUETOOTHPAN
Secondary PDP UMTS Driver	SPUD

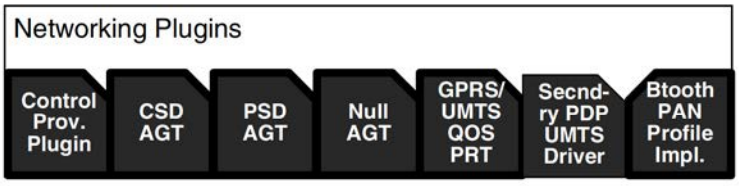


Figure 9.30 Networking Plug-ins

- The Connection Provider Plug-in provides IP connections to clients, supporting bearer mobility.
- The CSD AGT plug-in to the Connection Agent framework negotiates a circuit-switched data connection, for example to GSM or CDMA networks, supporting dial-up networking services.
- The PSD AGT plug-in is deprecated and its functionality is replaced by other components. It is an agent plug-in to the Connection Agent framework that negotiates packet-switched connection for example to GPRS networks, supporting 'always on' networking services.
- The NULL AGT plug-in implements a minimal agent used to pass straight through to an Ethernet connection that is provided by the Ethernet packet driver.
- The GPRS/UMTS QOS PRT is a plug-in helper module to the QoS Framework that gets and validates QoS parameters from the QoS framework at the request of a loaded NIF and is used to implement 3GPP parameters.
- The Bluetooth PAN Profile plug-in is an agent-like module that implements the Bluetooth Network Encapsulation Protocol (BNEP), as an Ethernet Packet Driver module. It serves as the network interface agent used to create PAN connections, enabling PAN to behave like a regular Internet access provider.
- The Secondary PDP context UMTS Driver (also called the PDP NIF) supports multiple primary PDP contexts (multi-homing over GPRS) on the telephony reference platform. It is not a production component.

Link Layer Control

Link-layer components of the networking stack, Network Interface modules (NIF files) are selected by the Network Controller and loaded, started and stopped by the Network Interface Manager to implement the interface to the physical link layer (which is, in turn, provided by networking device drivers, serial communications CSYs, or telephony TSYs). See Figure 9.31.

NIFs implement the polymorphic plug-in interface defined by the Network interface manager (NIFMan).

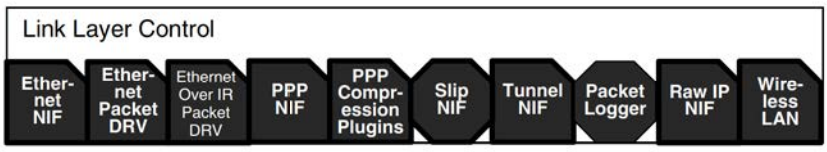


Figure 9.31 Link Layer Control components

Table 9.19 Link Layer Control Components

Component Name	Development Name
Ethernet NIF	`ETHER802`
Ethernet Packet Driver	`ETHERDRV`
Ethernet Over IR Packet Driver	`IRLANPACKETDRIVERS`
PPP NIF	`PPP`
PPP Compression Plug-ins	`PREDCOMP, MSCOMP, STACCOMP`
SLIP NIF	`SLIP`
Tunnel NIF	`TUNNELNIF`
Packet Logger	`PACKETLOGGER`
Raw IP NIF	`RAWIPNIF`
Wireless LAN	`802.11`

- The Ethernet NIF component provides a generic Ethernet layer network interface, that manages Ethernet framed packets. It is designed to sit below any number of supported Protocol modules and on top of more specialized Ethernet framing interfaces, called packet drivers.
- The Ethernet Packet Driver is an Ethernet framing interface, the driver-level component (DRV files, that is, lower-layer packet drivers) that supports the Ethernet NIF.
- The Ethernet Over IR Packet Driver is an Ethernet framing interface, the underlying networking interface driver for infrared.
- The Serial Line IP (SLIP) NIF component is supplied as a reference component that licensees can choose to remove or replace with a production implementation. SLIP was the earliest (and simplest) protocol for relaying IP packets over dial-up lines and has largely been replaced by PPP.
- The Point to Point protocol (PPP) NIF provides TCP/IP over serial communications (i.e. over a point-to-point link). It allows a device to connect to a phone and use it as a gateway to the Internet. Once the link has been established, optional facilities such as data compression may be negotiated.

- The PPP Compression Plug-ins supplies the implementation of common PPP compression algorithms as dynamically loaded DLLs. It includes Microsoft Compression (MSCOMP), Stac Electronics Compression (STACCOMP) and Predictor Compression (PREDCOMP) implementations.
- The Tunnel NIF component implements the IPSec tunnel to enable IPSec to operate in tunnel mode, for example, as used by VPN clients.
- Wireless LAN supports IEEE 802.11 wireless networking.

9.9 Short-link Services

Short-link services enable individual devices to communicate directly with each other ('peer-to-peer'), either over a physical cable connection such as serial or USB, or using short-range radio, either line-of-sight such as infrared, or unseen paired, such as Bluetooth. (Note that, by this definition, Wi-Fi, which is fast becoming important on phones, is considered a network access technology not a short-link connection technology, although Wi-Fi hardware supports a peer-to-peer mode.)

Symbian OS supports the principal short-link technologies: RS232 serial, USB, infrared/IrDA and Bluetooth, as well as the higher-level OBEX object transfer protocol, which is supported over both IrDA and Bluetooth.

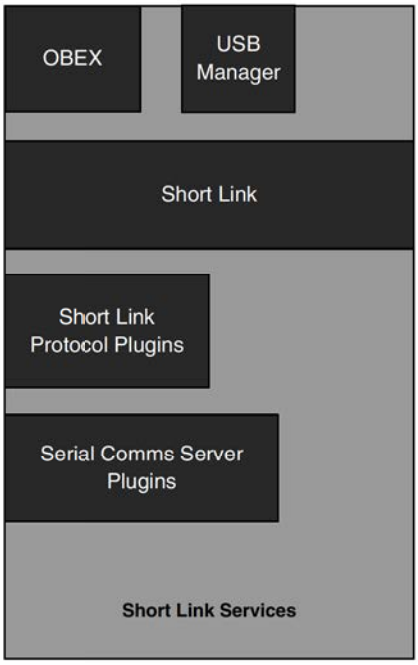


Figure 9.32 Short-link services

The short-link-services block includes managers, utilities, protocol implementations and serial-hardware-adaptation plug-ins. Associated device drivers are located lower down in the system model, at kernel level. See Figure 9.32.

For network-capable mobile devices (mobile phones and PDAs, for example), short-link connections are also important for network access. Typically, they provide the connection alternative to using the onboard phone. In Symbian OS, short-link services act as bearers for higher-level communications services, including both networking and telephony. This enables some interesting scenarios, for example, remote use of a phone in one Symbian OS device from another over a short-link connection.

Although continuing to evolve to enable increased data rates, short-link technologies are relatively mature and Symbian's support for them is relatively mature. RS232 serial has a long history and IrDA, Bluetooth and USB have all been standardized since the mid-1990s.

However, there are interesting and significant evolutions in all the technologies. In terms of connection speeds, while serial cable is limited to data transfer rates of 115 kbps, Bluetooth offers data rates closer to 1 mbps with a range of 10 meters, while 'newer', 'faster' IrDA standards increase rates beyond 16 mbps and even up to 100 mbps. USB began as a 12 mbps standard, before increasing 40-fold (with USB 2.0) to 480 mbps.

The application possibilities are also interesting and extend beyond basic data management and data synchronization. After a slow start, Bluetooth has become ubiquitous on phones, in particular for hands-free and headset peripherals, including stereo headsets. USB offers much more than just a physical link protocol. USB is both a link technology and a transport protocol definition with extras such as support for powering unpowered devices and hot-plugging ('plug and play' notification to the host). In a Symbian context, it allows a Symbian OS device to plug into a USB host (for example, a desktop computer) and offer multiple services.

Both IrDA and Bluetooth specify a complete protocol stack defining link, transport and application layers, which offers significantly more than just serial-like setup for a simple physical link.

Because Bluetooth allows ad hoc, 'promiscuous' connection between any devices within range, security is potentially an issue. The Bluetooth standard therefore includes security protocols (which Symbian OS implements).

As well as conventional serial communications, over a physical serial link or virtualized over IrDA or Bluetooth, Symbian OS supports a number of higher-level short-link services:

- Higher-level IrDA protocols are supported, for example including IrTranP for beaming camera images.
- IrDA Object Exchange (OBEX), a binary protocol for data exchange, is supported over IrDA, Bluetooth and USB connections.

- A number of Bluetooth profiles including security profiles are supported, with support for licensee extension.
- USB device management is supported.

Architecture

While short-link services forms a natural logical and functional block, it does not form a cohesive architectural unit. While the supported short-link technologies are designed to interoperate extensively and implement the overall architectural patterns of communications services (server- and framework-based, protocol module plug-ins to the Socket Server, serial port plug-in implementations to the Serial Server framework), the detailed architecture of each is distinct and should be understood independently of the serial architecture.

IrDA is implemented as a Socket Server plug-in module, loaded by the Socket Server when an IrDA socket is requested (either directly by an application, or by other components in the Comms Services). Within the Socket Server session, the protocol module communicates with the infrared port through the Serial Server and its IrDA serial plug-in CSY module, which ultimately drives the logical and physical device drivers for the onboard infrared hardware.

The OBEX implementation is designed as a wrapper for either a socket style API (RSocket for IrDA and Bluetooth) or a USB client API (RDevUsbcClient for USB). OBEX is implemented as a static DLL to which clients link at compile time, with the OBEX code running in the client thread.

Bluetooth is implemented as a Socket Server protocol plug-in module. Clients request a Bluetooth socket from a Socket Server session. The Bluetooth socket communicates with the firmware controller via the Bluetooth HCI implementation. Symbian OS implements the mandated v1.2 Bluetooth stack.

IrDA and OBEX

Symbian OS has supported IrDA since the first ER5 release, providing line-of-sight infrared data exchange between devices. IrDA is more than a simple connection protocol and, in fact, comprises a complete set of protocols from application level to link level, including IrTranP (Infra Red Transfer Picture, for devices with cameras), IrCOMM (IrDA serial port emulation) and TinyTP (TinyTransfer Protocol, providing flow control), as well as lower-level protocols including FIR (Fast Infrared). All are supported by Symbian OS.

IrDA also provides the underlying support for OBEX over infrared (Infrared Object Exchange, IrOBEX). OBEX is a protocol and not a service but application-level services can be created that use the protocol to

send and receive data. At the application level, Symbian OS provides OBEX-based services including SendAs messaging, SyncML data synchronization, installer services, and so on. Symbian OS has supported OBEX since the first ER5 release. Since the introduction of Bluetooth support in Symbian OS v6, it has supported OBEX over Bluetooth and, since Symbian OS v7, OBEX over USB (but with server functionality only).

Bluetooth

Bluetooth also defines a complete protocol stack and not just a radio link technology. The Bluetooth services that run on top of the stack are defined as Bluetooth profiles. Symbian OS provides Serial Port, PAN (Personal Area Networking) and Generic Access profiles, as well as Remote Control (since Symbian OS v9), that enables a Symbian device to control Bluetooth peripherals, for example headsets. Licensees may add additional profile support.

Bluetooth components include:

- The Bluetooth Manager is the information store (implemented over Symbian OS DBMS) used to manage details of local and remote Bluetooth devices.
- Bluetooth SDP (Service Discovery Protocol) enables Bluetooth devices to find each other and store information about discovered devices. (The SDP database is not persistent.)
- The Bluetooth HCI (Host Controller Interface) interfaces the Bluetooth stack to the onboard controller hardware and is provided as a reference plug-in.

Symbian OS has supported Bluetooth since Symbian OS v6, with incremental support added over subsequent releases.

USB Manager and Classes

USB classes are analogous to Bluetooth profiles and represent the use cases that a device supports when it connects to a USB host. The USB Manager on a device enumerates, starts and stops the USB classes implemented on the device and provides a query interface for their status, providing a central control point and an on–off switch.

Symbian OS provides a USB Manager and implements USB CSY (serial over USB), Mass Storage and OBEX (OBEX over USB) classes. The USB Manager implements a server interface for USB class implementations and for clients requesting information or services from USB classes (typically the user is the USB host) and provides the underlying mechanism for application-level class configuration and querying of the USB host (the other connected device) across a USB connection.

Component Collections

OBEX Collection

This collection defines the OBEX (Object Exchange) session protocol. OBEX is a binary protocol and is therefore compact and can support application-level services from simple beaming of vCard and vCal entries to full-scale synchronization, for example, as a SyncML bearer.

Table 9.20 OBEX Components

Component Name	Development Name
OBEX Protocol	`OBEX, IROBEX`
OBEX Extension API	`OBEX_EXTENSIONAPIS`

In Symbian OS, OBEX is supported over IrDA infrared, Bluetooth and USB, providing session-style APIs, that is, Connect and Disconnect and basic Get and Put commands. See Figure 9.33.

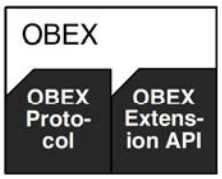


Figure 9.33 OBEX components

USB Manager

This collection comprises the manager for the USB classes present on a device, for example providing the mechanism beneath a configuration application like a control panel to switch on and off the available USB classes on a Symbian OS device and to query a USB host (not a Symbian OS device) application across a USB connection. See Figure 9.34.

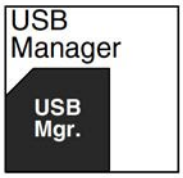


Figure 9.34 USB Manager components

Table 9.21 USB Manager Components

Component Name	Development Name
USB Manager	`USB`

Short Link Collection

These higher-level components support the Bluetooth protocol implementation and Bluetooth profiles. See Figure 9.35.

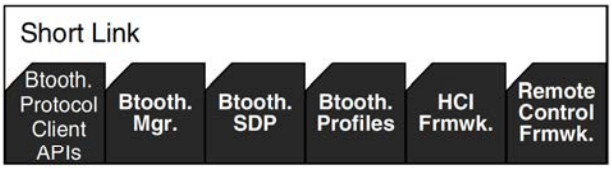


Figure 9.35 Short Link components

Table 9.22 Short Link Components

Component Name	Development Name
Bluetooth Protocol Client APIs	`No unit`
Bluetooth Manager	`BLUETOOTHMANAGER, BLUETOOTHBTEXTNOTIFIERS, BLUETOOTHCONFIG, BLUETOOTHGAVDP, BLUETOOTHROM, BLUETOOTHUSER`
Bluetooth SDP	`BLUETOOTHSDP`
Bluetooth Profiles	`BLUETOOTHAVRCP`
Remote Control Framework	`BLUETOOTHREMOTECONTROL`
HCI Framework	`BLUETOOTHHCI`

- The Bluetooth Protocol Client APIs are used by Bluetooth socket clients and provide support for low-level control of protocol parameters (packet sizes, for example) and hardware (power modes, for example).
- The Bluetooth Manager provides an information store for managing details of the local and remote Bluetooth devices, implemented over Symbian OS DBMS, allowing entries to be stored, retrieved, modified and deleted.

- The Bluetooth Service Discovery Protocol (SDP) is the mechanism used by connected Bluetooth devices to query each other and exchange information about the Bluetooth services they support.
- The Bluetooth Profiles include Generic Access Profile (GAP), Personal Area Networking (PAN), since Symbian OS v8, and (from Symbian OS v9) Audio and Video Remote Control (AVRCP).
- The Remote Control Framework enables sending and receiving of remote-control commands to and from remote Bluetooth devices. (It is supported from Symbian OS v9.)
- The HCI Framework is a reference implementation of the Bluetooth Host Controller Interface as used by the Bluetooth Stack to interface to the onboard controller hardware. It provides a full range of HCI commands, accessed indirectly via L2CAP and RFComm layers. Licensees can replace the supplied implementation.

Short Link Protocol Plug-ins

This collection implements the Bluetooth core stack, including the Bluetooth protocols and the HCI firmware implementation and the IrDA protocol suite as PRT Socket Server plug-in-in protocol modules. See Figure 9.36.

Table 9.23 Short Link Protocol Plug-ins

Component Name	Development Name
Bluetooth Stack PRT	`BLUETOOTHSTACK`
Bluetooth HCI	`BLUETOOTHHCIPROXY`
IrDA PRT	`IRDA, INFRA-REDCONFIG`

- The Bluetooth Stack PRT component implements the Bluetooth stack as a Socket Server protocol plug-in, providing a complete implementation including L2CAP, RFCOMM and SDP.
- The Bluetooth HCI is a reference implementation of firmware-specific support for the standard Bluetooth Host Controller Interface (the

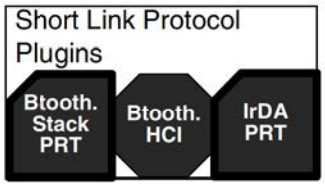


Figure 9.36 Short Link Protocol Plug-ins

stack-side implementation of the interface forms part of the standard Bluetooth support provided by Symbian OS).

- The IrDA PRT is an implementation of the IrDA protocol stack as a Socket Server protocol plug-in, provides a complete IrDA implementation including IrTranP (for sending pictures) and FIR (Fast Infrared).

Serial Comms Server Plug-ins Collection

CSY modules are implementations of serial ports virtualized over different bearers (RS232, USB, Bluetooth, IrDA) and are loaded by the C32 Serial Server in response to clients to provide ports of the types requested. See Figure 9.37.

Table 9.24 Serial Comms Server Plug-ins Components

Component Name	Development Name
Serial Port CSY	ECUART
USB CSY	ECACM
Bluetooth CSY	BTCOMM
IrDA CSY	IRCOMM

The Serial and IrDA CSY components were both present in ER5.

- The Serial Port CSY component implements an RS232 virtual serial-port abstraction for conventional serial communications and directly drives the ECOMM.LDD and ECOMM.PDD logical and physical device drivers.
- The USB CSY component was introduced in Symbian OS v7.0 supporting a single-port configuration and extended to support multiple virtual ports in Symbian OS v7.0s. It provides a multiple serial-port-like interface over a USB connection and directly drives the EUSBC.LDD and EUSBC.PDD logical and physical device drivers. Note that this is an implementation of USB intended for legacy applications that require conventional serial support, rather than for USB-aware applications.

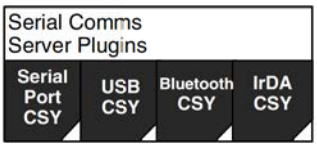


Figure 9.37 Serial Comms Server Plug-ins

- The Bluetooth CSY component was introduced in Symbian OS v6.1 with the first Bluetooth implementation for Symbian OS. It is a plug-in to C32 Serial Server and implements an RS232-like virtual serial port over a Bluetooth link using an RFComm socket. Port configuration is performed using the Bluetooth Manager APIs.
- The IrDA CSY component implements the IrDA standard for serial communications, IrComm, emulating a serial port over an IrDA link. Internally, it uses an IrDA socket (IrDA.PRT), through a Socket Server session, which in turn drives the ECUART.LDD and UCUART.PDD logical and physical drivers to drive the infrared hardware.

10

The Base Services Layer

10.1 Introduction

To get Symbian OS up and running on new hardware, whether on a reference board (from a supplier such as Intel or Texas Instruments) or on the hardware for a new phone, you need to port the base layers of the system.

The lowest level of the system contains the operating system kernel, device drivers, and the device-driver framework support, which provide operating system primitives and hardware abstraction frameworks. Sitting just above them are the low-level libraries, servers, and frameworks that build on the kernel layer to create a programmable and usable operating system. Because Symbian OS is a microkernel system,[1] the 'kernel side', which runs in protected or privileged mode on the host processor ('supervisor' mode on ARM processors), is kept as small as possible. The kernel-side/user-side distinction roughly divides the base of the system into two layers.

The Base Services layer is the higher of the two layers and it contains the user-side servers, frameworks, libraries and utilities that build on the kernel layer to provide the basic operating system services. Together, the two layers constitute the minimal system which can be booted, run and programmed on real hardware. In a monolithic operating-system design, most (and possibly all) of the Base Services would form part of the kernel implementation. See Figure 10.1.

10.2 Purpose

The Base Services layer extends the bare kernel into a basic software platform that provides the foundation for the remaining operating system

[1] In fact the design is not 'pure' microkernel, but borrows from both microkernel and monolithic design principles (see Chapter 11).

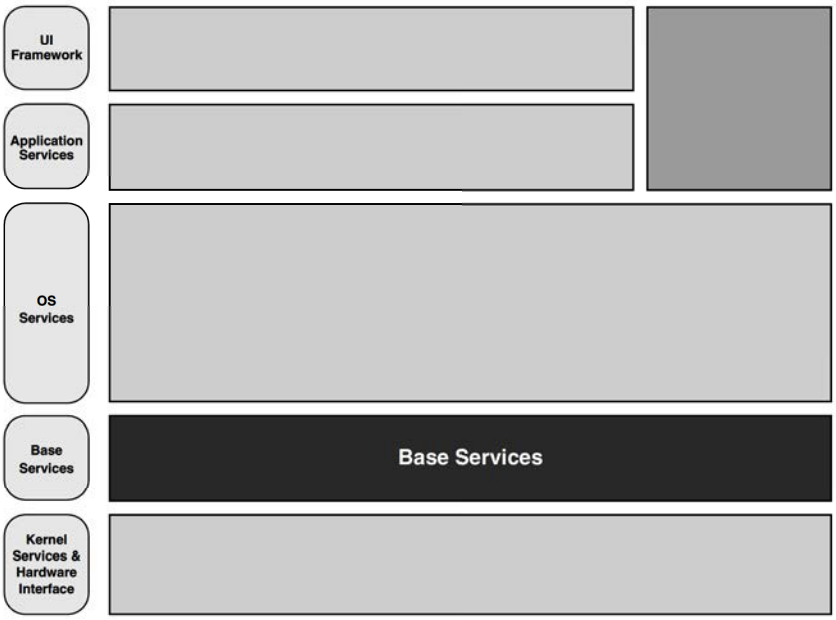


Figure 10.1 Base Services layer in the system model components

services, and effectively encapsulates the user side of the 'base' operating system. It also provides the minimum services required to enable a complete and self-contained basic build of the lower-level system, which supports only text-mode program execution and is used to create the first stage 'base-port' to new hardware.

As well as providing foundational frameworks and utilities which are used both by system components and by applications, it also provides the operating system libraries that support the programming model, in other words, which support the creation, loading, and running of programs on the operating system and which implement many of the signature Symbian OS idioms, for example the cleanup stack, active objects and descriptors.

10.3 Design Goals

In many (but not all) respects, Symbian OS offers a textbook example of a microkernel operating system architecture.[2] The most significant exception is the inclusion of the two-level device-driver framework, and

[2] See the rationale for and description of the microkernel pattern in [Buschmann *et al.* 1998].

device drivers themselves, on the 'kernel side' of the system. A true microkernel design would move these into user space.

The microkernel principle is to keep the kernel small;[3] core functionality which is, however, above the level of the basic operating system primitives, is kept out of the kernel itself and instead is located in system servers. System servers extend the microkernel to provide necessary services, and also encapsulate any lower-level software and hardware dependencies. In Symbian OS, the core system servers that are required to create a complete but minimal running system on real hardware are located in the Base Services layer; the remaining system servers, which are not essential for a basic hardware port but which are required to engineer a complete product based on Symbian OS, are located one layer up, in the OS Services layer.

The goals of the Base Services layer therefore are to provide efficient and effective extensions to the basic kernel functionality, which are in a concrete sense complete (i.e. they enable a complete but minimal system to be built), while being both portable and extensible.

10.4 Overview

The Base Services layer includes a number of essential frameworks and libraries on which almost all higher-level services, as well as applications, have some direct or indirect dependencies.

- The User Library provides the basic programming model for Symbian OS, including system-specific types (such as the `CBase` class and manifest constant[4] definitions), as well as the APIs that define the unique native idioms, for example active objects, descriptors and UIDs, libraries which provide DLL and executable entry point stub classes, and so on.
- The File Server includes file-system utilities and the concrete file-system implementation plug-ins in use on a particular device.
- The Store is a persistent storage framework. The Base Services layer also includes the DBMS implementation, as well as more recent additions such as the Central Repository, which provides a single location and set of APIs for managing all system settings.

[3] The most significant immediate benefits of 'small' are portability, because all essential hardware dependencies are encapsulated within the small core of the system, and small memory footprint, a small system consuming less ROM (where the system is ROM-based) and RAM (put simply, there is less system to load at runtime). The additional goal of simplicity is also more likely to be realized in a small system than in a large one.

[4] Those are named constants whose underlying definition can be varied at compile time for different platforms; in Symbian OS, they include `TInt`, `TReal`, `TBool` and `TAny`.

- Other essential frameworks and libraries include the Plug-in Framework (ECOM), cryptographic libraries, Application Utilities (such as the Basic Application Framework Library, BAFL), character encoding and conversion libraries, XML parsers,[5] the power management and shutdown framework, as well as the low-level framework support used by multimedia services to communicate with hardware-accelerator adaptor plug-ins. (The actual adaptors and the device drivers with which they interact are located lower down, at the kernel level.)
- Components such as the Text Window Server and Text Shell are required to make the base system complete and to avoid dependencies on higher-level services, for example, graphics.

Put simply, from a programming perspective, many of the most basic characteristics of the operating system are realized in the Base Services layer.

10.5 Architecture

The Base Services layer of Symbian OS is in many ways the foundational layer of the system, extending the microkernel and the lowest level hardware-abstraction services provided by the kernel layer into a basic but complete system. A number of critical services which in monolithic architectures would be included in the kernel itself, for example the file system and the user libraries which provide the programming model for the operating system, are found here. The key boundary which defines the separation of these services from the kernel is the division between kernel (supervisor or privileged) and user (non-privileged) processes. In a monolithic system, most of these services would run as privileged kernel processes.

The design decision to separate these services from the kernel and to implement them as user-side services is a distinguishing feature of the operating system, separating it from monolithic systems (Unix/Linux, Windows-derived systems) and putting it squarely in the tradition of microkernel operating system design.

From the perspective of applications and higher-level operating system services, the Base Services layer libraries and frameworks provide the logical interface to the basic low-level operating system. The Base Services layer extends the raw hardware support and the basic kernel abstractions of the low-level system and adds file-system support and the File Server,

[5] XML is considered an essential service since XML is increasingly used as the basis for internal configuration files and other essential data formats, for example, Central Repository entries.

the User Libraries that support the programming model, a simple text-window server and a text-based shell, and an assortment of other low-level frameworks and utilities. Together, this is enough to support, test and validate a first-stage port to new hardware and it provides the foundation for creating complete support for all device hardware. The boundary between the Base Services layer and the higher-level services in the layers above it, therefore, is a concrete one: nothing above the Base Services layer is required get a port running on specific hardware.

The system model organizes the Base Services layer components into a number of collections, divided broadly between the low-level components that interact closely with the kernel to provide basic services (the User Library, file-system support) and higher-level components that build on these services (for example, Store, which provides the persistence model, the Cryptography Library and the Text Shell).

The User Library

It is through the User Library that the fundamental abstractions implemented by the kernel, which together define the native programming model for Symbian OS, are made available to clients. These include processes, threads and memory chunks and mutexes, semaphores and message queues. The User Library also implements many other programming idioms specific to Symbian OS, including active objects and descriptors, the cleanup stack, the client–server framework, and the Publish-and-Subscribe mechanism. It supplies an assortment of utility classes, including timers, date and time services and locale definition and collection classes, including arrays, lists and binary trees. It defines the native data types, both class-based and manifest constants, and supplies the libraries that implement the low-level system and language bindings, including DLL and executable entry point stub classes.

In the original kernel architecture of Symbian OS (EKA1, before Symbian OS v9), the User Library was called from both user-side and kernel-side code. In order to guarantee time bounds, the EKA2 kernel-side code does not link to the User Library but instead uses a small utility library (incompatible with the user-side library) accessible only by the kernel side.

The User Library includes the following APIs:

- the native types used in the system which include C++ base classes (including `CBase`) and manifest constants (`TInt` and others)
- collection classes (buffers, arrays and lists), descriptors, Unicode-character support, raw-memory management (copying and filling) and geometric concepts (points, sizes, rectangles and regions)
- math libraries including 64-bit integers and floating-point math

- idioms specific to Symbian OS including the cleanup stack, descriptors, active objects, UID manipulation, and implementations of memory allocators, named and reference counted objects and bitmap allocators
- other useful classes supporting lexical analysis, bitstreams, Huffman compression, timers and timing services.

In addition, it supplies libraries that provide DLL global data and static data and thread local storage; and executable and DLL entry point support (for example calling static constructors).

The User Library also provides the Publish-and-Subscribe mechanism (since Symbian OS v8), as a means of storing system-wide global variables and a platform-security safe IPC mechanism (again, since Symbian OS v8) for peer to peer communication between threads in the operating system. Publish and Subscribe is based on the notions of properties (data values), publishers (threads with rights to update given properties), and Subscribers (threads interested in changes to given properties). Because it is available on both user-side and kernel-side, it also provides a possible asynchronous communication mechanism between user-side and kernel-side code.

The File Server

The File Server provides the framework architecture supporting the implementation of file systems as custom plug-ins and the default plug-in implementations for FAT file systems, the native format for externally visible drives, for example, those implemented on removable media, as well as internal-only formats such as Read Only File System (ROFS), the internal file system to which ROM code is copied for execution in hardware architectures that do not support execute-in-place memory.

File-system plug-in implementations may in turn be further extended via extension DLLs to support specific hardware differences, for example FAT on NAND flash, which implements a NAND flash translation layer transforming requests coming from the FAT file system into a format suitable for a NAND flash-media driver. Note that the file server is multithreaded (since Symbian OS v9), using one thread per storage medium used.

The File Server also provides some file-related utility functions, for example FAT filename conversion which supports translation from full Unicode file names to ASCII. (While EKA1 supported Unicode strings internally, the real-time EKA2 kernel uses only ASCII strings internally; note that there is no impact on the full, system-wide support for Unicode.)

The file server has traditionally had an additional role in Symbian OS, as the first of the system services to be started by the final stage of the kernel boot process. In Symbian OS v6 and v7, the file server was responsible

for starting the Window Server, in effect completing the boot process. From Symbian OS v8, the File Server instead launches the System Starter, which performs final initialization of the File Server including adding and mounting file systems on appropriate local drives, and then initiates start-up of the rest of the system, including implementing the customizable server start-up policy (which defines which servers should be started and in which order).

Essential System Frameworks

The Base Services layer includes some essential system frameworks, including the Plug-in Framework, which underpins the Symbian OS framework–plug-in architecture, and the persistent storage model.

Plug-in framework

The Plug-in Framework, known as ECOM, has two principal purposes: to make it easier to design and implement new services or features as framework plug-ins by providing a standard (and best-practice) pattern together with ready-made run-time support. Framework plug-in architectures improve the overall modularity, extensibility, and customizability of the system, thus improving usability (from a system perspective) as well as improving design consistency. As importantly, it provides an evolution path for already conforming framework plug-in components to migrate relatively painlessly to the platform security model introduced in Symbian OS v9, making it easier for components to adopt the required security policies (i.e. to ensure trust between frameworks and the plug-ins they load and to avoid plug-in loading being exploited to subvert platform security).

ECOM defines an interface to which all plug-ins conform (plug-ins derive from the ECOM base classes) and provides the dynamic discovery and instantiation mechanisms which find, create, and load them on demand.

ECOM's original design was evolved from the design of the WAP browser framework plug-ins. Broadly, it provides:

- methods for defining and implementing interfaces as DLL plug-ins
- plug-in registration and methods for managing multiple interface implementations, including plug-in 'upgrades' (later versions)
- fast dynamic discovery and instantiation methods for plug-ins, as well as static registration for known system plug-ins
- capability policing, that is, enforcement of the security restrictions of its clients

- other features including support for easy localization of plug-ins and start-up state awareness (to improve system boot-up performance).

ECOM was first introduced in Symbian OS v7 and was then significantly enhanced in Symbian OS v8, to support and conform with the new platform security model. Initially it offered an optional, standard mechanism for frameworks to define plug-in interfaces and a standard plug-in registration and loading mechanism. Subsequently it was elevated from an optional to an obligatory mechanism; from Symbian OS v8, it is the standard interface used by all frameworks to define how plug-ins interact with and extend the framework and the global runtime binding mechanism that finds and loads plug-ins into requesting frameworks on demand, while conforming to the Platform Security requirements and limitations on processes.

Security issues

ECOM ensures that frameworks are only able to find plug-ins they have the capability to load and which pass the platform security check, that is, matching of the DLL UID field from the RSC resource file to the SID (secure identifier) of the corresponding DLL.[6] Plug-ins are loaded into the requesting client framework's process, allowing the kernel to police the capabilities of the plug-in DLL. (Although if the plug-in's capabilities do not match those of the client process, then it could be loaded into a separate process.)

ECOM is implemented with a standard client–server architecture, based around a central registry (of interface implementations) and a server client-side API that handles inter-process communication (IPC) between servers and their clients (wrapping the invocation parameters, passing the wrapped request over the IPC boundary and unwrapping any return parameters when a call completes). Client frameworks use a session object as the interface to the ECOM server for finding, creating and destroying plug-in providers of the framework interface.

Calls to the ECOM server are translated into registry or load calls to perform:

- addition and removal of interface implementations (registrar functions)
- access and persistence mechanisms (registry data functions)

[6] Strictly speaking the UID3 of a DLL is not really the SID, since SIDs are only assigned to executables or processes (based on the executable's UID3) and not to DLLs. Also any single DLL can potentially contain multiple different implementations of a given interface, which would share interface UIDs but differ in implementation UIDs. [Heath 2006] is the best reference for following up the details.

- resolution and searching mechanisms returning 'best fit' results (resolver functions)
- loading and unloading (load manager functions).

A single instance of the registry exists. Registry data is held in two forms, an internal format for fast access, consisting of a subset of the full registry data, and persisted data, consisting of the registration set stored in file form, divided into branches with one branch per available drive (branches may be transient, supporting removable media).

Client frameworks (i.e. interface definers) may supply custom resolver implementations to ECOM to implement custom criteria.

Full discovery of plug-ins occurs at ECOM server start-up, that is, at device boot time. Additional discovery of non-read-only internal drives occurs when a drive is added or removed and when a secure plug-in is added to or removed from a writable drive.

Persistence model

The Symbian OS persistence model is based on the Store architecture, which defines abstractions of streams and stores.

A stream is an abstract interface that translates between internal and external object representations, that is, between bit layouts in RAM and bit layouts saved onto storage media or sent over a network. As well as encryption and decryption streams, four alternative stream implementations are provided, suited for different underlying storage media:

- fixed-size memory streams
- variable-size memory streams
- file streams
- store streams.

A store is an abstract interface that allows a network of streams to be manipulated, including Externalize and Internalize operations, which allow complex data structures (e.g. whole documents or databases) to be stored or restored from external media or from a network.

As well as secure stores (which provide encryption and decryption) and supporting store dictionaries (used to locate the various streams inside a store), the Store architecture provides alternative implementations suited for different underlying storage media or uses:

- stores using RAM as the underlying storage media (for example, used as undo buffers by some applications)

- stores using files as underlying storage media, either *direct* file stores used by 'file-based applications', which keep all their data in RAM when running (in other words, which create and manipulate conventional documents), or *permanent* file stores used by applications that only part-load their data (for example, database applications)
- stores that can be embedded into other stores thus allowing document embedding to create compound documents (e.g. pictures in a text document)

Streams and stores provide the native, object-oriented persistence model for Symbian OS. Both the DBMS relational database interface and the Central Repository are implemented on top of store mechanisms.

DBMS

The DBMS component defines a general relational-database-access API and provides implementations either for small client-side databases or for client–server-based multiple-client implementations. Client–server databases are stored in files. Client-side databases can either be a whole file or a single file stream (enabling multiple single stream databases to reside in a single file).

Databases can be manipulated either through a native API or a subset of SQL. Basic database functions are supported, including table creation, manipulation and deletion, database queries and transactions.

From Symbian OS v8, where required, DBMS supports security-access-control policies for databases, including shared-access policies. For system-supplied databases, it allows additional finer-grained policies to be specified for named tables within a database (for databases created within the DBMS private data-cage).

Central repository

The Central Repository provides a single persistent store for global settings as well as a notification mechanism allowing clients to register to be notified when specific settings change.

The Central Repository is designed as a collection of repositories, where a repository is a collection of settings. A setting is represented by a data value (a 32-bit integer, a real number, a byte-array or a text string). Repositories are created from a definition file based on a standard template and may be compacted into a binary format. Each repository has an owner and is required to declare an access-control policy, which is set in the initialization file and cannot thereafter be changed. Access control may be specified at the level of the whole repository, for individual settings or for ranges of settings, and may include settings which have not yet been created.

Depending on access control, individual settings may be created, searched for, have their values set, or deleted. Range operations are supported and a notification registration mechanism is provided allowing clients to register interest in settings changes (including creation and deletion). 'After-market' repositories (e.g., for user- or network-installed applications) are supported by the Application Installer. Backup, restore and caching of repositories is also supported. Access to repository settings is restricted based on the capabilities of the client making an access request together with the repository security policy.

In general, settings replace the use of INI files to store application and system defaults and other information, for example default file names, locale settings and user preferences. Similarly, settings replace the use of the Comms Database for storing communications-specific defaults and settings, although the Comms Database interface is preserved for compatibility.

The earliest releases of Symbian OS included a Registry, but it was removed (as it was not portable) in Symbian OS v6 and replaced by solutions based on INI files and the Comms database. In Symbian OS v8, the Central Repository was introduced to provide more efficient and consistent settings management.

Other Services and Utilities

The Base Services layer contains a number of additional frameworks, libraries, utilities and servers.

Application Utilities

The Application Utilities, known to developers as the Basic Application Framework Library (BAFL), provide an assortment of utility classes organized as a single library DLL:

- resource-file handling including loading and reading of legacy formats (before Symbian OS v7) and Unicode-compressed and Unicode-and-dictionary-compressed formats (since Symbian OS v7), including robust reading classes able to handle corrupt resource files
- file utilities, including file finding based on file type as defined by UID and file matching to select between files based on the current locale
- string pools, a storage mechanism allowing for fast string comparisons
- dynamic arrays for descriptors, supporting mixed 8-bit and 16-bit descriptors
- incremental text-matching comparing two text buffers (reading left-to-right)

- support for showing localized names of 'user-showable' plug-ins
- clipboard copy–paste support implemented as a direct file store with stream dictionary, allowing applications to retrieve clipboard data by UID
- system sounds for messages, events, errors and so on, specified by UID
- minimal support for spreadsheet-style 'cell' and 'range' data types
- legacy change notifier (derived from active objects) wrapping the `RChangeNotifier` for system environment changes relating to time, locale, power and thread death.

Character Encoding and Conversion Framework and Plug-ins

The Character Encoding and Conversion Framework provides an API for converting text between Unicode and other character sets based on an extensible converter plug-in architecture.

In Symbian OS v9, conversion is supported for a variety of ASCII formats (including common ISO codepages), UTF-7 encodings (including Shift-JIS and JIS) and UTF-8 encodings. Conversion is performed by specifying the Unicode character set of interest (for conversion to or from) and then requesting the conversion.

As well as text conversion, text utilities are provided to manage character sets (create character-set arrays, find the character-set UID from the character set name and vice versa) and to detect character sets automatically based on sample texts.

XML Framework and Parser Plug-ins

The XML Framework provides an extensible framework for XML parsing based on a parsing model similar to SAX 2.0, into which custom parser-implementation plug-ins (as well as validator, DTD and auto-correction plug-ins) can be loaded. Default plug-ins are provided for non-validated parsing of XML 1.0 and for WAP Binary XML (WBXML).

Parsers are selected based on a document's MIME type and other criteria supplied by clients when using the framework. The parser class defines methods that parse XML data from descriptors (all in one go or incrementally) and from files. Internally within the parser, text is stored in UTF-8 format to ensure preservation of extended characters.

The WBXML parser plug-in can be extended to support additional document types by providing WBXML token-to-string translation tables ('String Dictionaries'). Default tables are supplied for SyncML, WML and Service Indication.

The design goal for the framework is to provide a single, standard, platform implementation of a flexible and capable XML parser to replace

the various task-specific and ad hoc parsers provided locally in the system. The framework also provides sufficient extensibility for likely future uses (including generating capability).

So-called 'processor' plug-ins (for example, validators and auto-correctors) may be chained with parsers to provide multiple processing stages.

String Dictionaries are implemented using string pools that make string comparison almost instantaneous (at the expense of string creation; however, this supports parsing cases where string constants are known at compile time particularly well, as is the case where documents follow standard DTDs such as SyncML, SMIL or WML).

Media Device Framework and Plug-ins

The Media Device Framework provides hardware-abstraction interfaces for audio and video accelerators to the Multimedia Framework (see Chapter 8) and its clients. Typically, accelerators are hardware devices (codecs) but they may also be software emulations. The framework defines APIs for sound, video, MIDI, and ASR (Automatic Speech Recognition) accelerators, and the architecture for loading the lower-level adaptor plug-ins (DevSound, DevVideo, DevMIDI, and DevASR; see Chapter 11). A client utility API for speaker-independent speech recognition is also supplied as a plug-in and is available to any client wanting to interface to ASR hardware (or software emulations).

The framework also includes a policy server that manages access to the underlying audio and video hardware, deciding which clients can access the hardware and when. Licensees can customize access policies.

The Media Device Framework evolved from the earlier Symbian OS v6 and Symbian OS v7 MediaServer. Previously, Multimedia Framework controller plug-ins were able to directly interface to audio and video codecs via adaptor plug-ins. By defining a standard interface between controllers and adaptors, the Media Device Framework enables portable adaptors to be developed to support specific accelerator hardware. The framework has evolved significantly over subsequent releases compared with its first implementations, which supported only audio.

Cryptography Library

The Cryptography Library provides system-level support for a wide-range of non-RSA cryptographic algorithms including symmetric and asymmetric ciphers, hash functions and a cryptographic strong random-number generator. The cryptographic algorithms are supplied in two variants: export-restricted (strong) and non-export-restricted (weak). Note the change since Symbian OS v7, which provided an export-restricted and an RSA-based library, with no non-export-restricted variant.

Subcomponents of the library include:

- random-number server, an implementation of a random-number generator
- random-number library DLL, providing an API for generation of cryptographically strong random numbers
- hash library DLL, providing an API for generating cryptographic hashes, supporting MD2, MD5, SHA1 and HMAC
- password encryption API DLL supporting key generation from password (PKCS#5 key-derivation function) and key-based encryption and decryption
- cryptographic library, providing non-RSA cryptographic algorithms, supplied in weak and strong versions (depending on possible export restrictions) and implementing symmetric and asymmetric ciphers, padding schemes, and big integers.

Clients link against the Cryptography Library for all functions. Calls are transparently forwarded to whichever version of the library implementation is present at run time (strong if present, weak if not; this is determined at ROM build time). The weak version is limited to symmetric cryptographic operations with a maximum key size of 56 bits and asymmetric cryptographic operations with a maximum key size of 512 bits.

Zip Compression Library

Port of the zlib compression library (see relevant RFCs, for example, RFC1950) used to support compression and decompression of SIS files (Symbian native installable-application format) and Java Archive (.JAR) files, and for PNG decompression.

Shutdown Server

The Shutdown Server provides a notification service to clients to provide 'save data' and 'release resources' notifications in case of switch-off or low memory and similar events, enabling a client to save data (for example, if it is an application) and possibly also close itself (to free up resources).

It consists of a client-side library that clients use to request notifications, the Shutdown Server that provides 'save data' notifications and which may be derived from to create bespoke shutdown servers (for example, Uikon implements a customized shutdown server), and a server launcher (executable) that launches the service.

Feature Registry

The Feature Registry (introduced in Symbian OS v9.2) provides an API enabling run-time queries to discover whether known but optional features are supported on the particular running platform (device or emulator).

A 'feature' is a Symbian OS or user interface variant API (or set of APIs) identified by a Feature UID.

A configuration file listing features present is generated at ROM build time (on real devices) or provided as part of the emulator support in the licensee SDK and is held in a Publish and Subscribe property queried by the Query API. A Notify API is also provided but not currently enabled, with the intention that in future releases the feature set will be updatable at run time (the Symbian OS v9.2 implementation fixes the feature set at ROM build time).

Text Shell and Text-Window Server

Together, the Text Shell and the Text-Window Server that supports it make the base layers of the operating system independent of higher-level services (for example graphics and windowing support as well as the GUI-based application support), allowing functional text-mode-only builds of the base to support porting and other low-level development.

This enables a minimal but functional system to be built for and run on new hardware as a first step to providing full hardware support. In principle, all hardware dependencies are encapsulated within the base layers of the system; once the base port is complete, the rest of the system can be moved over to run on top of it without any further adaptation being required. In practice, the situation is a little more complex; Comms Services, in particular, are hardware-dependent at the lowest level of hardware abstraction and interface. In practice therefore porting is a two-stage activity: once the base port is complete, a communications port is needed to interface the communications stacks to the device hardware. When the communications port is complete, the remaining system services can be moved over.

The Text Shell provides a console-like (command-line) interface to basic operating system services, for example navigating the file system and launching executables, when standard graphics, application, and GUI support are not available. The Text Shell is also available on the emulator, where it is used, for example, when developing servers that run without a user interface.

The Text-Window Server supports the Text Shell, using a text-mode display driver to provide standard VGA/LCD screen displays on local hardware as well as VT100 terminal emulation over a serial line.

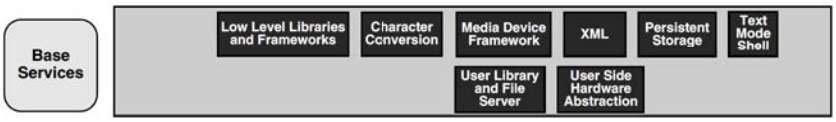


Figure 10.2 Component collections in the Base Services layer

10.6 Component Collections

The Base Services layer (see Figure 10.2) contains several collections of components.

- The User Library and File Server and User-Side Hardware Abstraction collections contain essential system services providing file-system support and essential user libraries.
- The Text-Mode Shell provides character-based text services that enable the lowest two layers of the system to be built independently of graphics frameworks.
- The Low-Level Libraries and Frameworks, Character Conversion, Persistent Storage and XML collections contain frameworks and libraries useful to applications, as well as to other system components.

User Library and File Server Collection

The User Library and the File Server implement essential basic functionality that should be considered central to the operating system. They interface to the kernel in a uniform way using the standard client–server model. Because they run user-side, the kernel is protected both from programming errors by users of the basic libraries (including resource exhaustion) and from timing latencies introduced on the user side enabling real-time guarantees to be met. See Figure 10.3.

- The User Library component provides much of the signature functionality of Symbian OS to (system) programs and to applications,

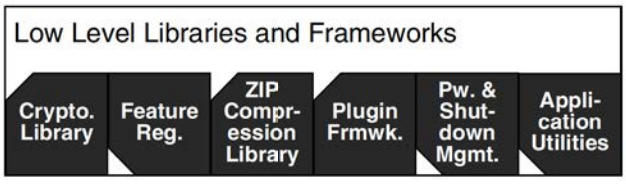


Figure 10.3 User Library and File Server components

Table 10.1 User Library and File Server Components

Component Name	Development Name
User Library	EUSER
File Server	F32_EKA2
Filesystem Plug-ins	FILSYS
FAT Filename Conversion Plug-ins	FATCHARSETCONV

including native data types, clean up and clean-up-aware base classes, active objects, descriptors, as well as the system–language binding including DLL stub mechanisms, IPC and similar mechanisms, and generally useful low-level services including (since Symbian OS v8) Publish and Subscribe.

- The File Server component manages all file access through client-side file-server sessions. It is a framework of file-system plug-ins and extensions which supports the implementation of custom file systems. The server is responsible for brokering client requests and passing them through to the file system, where the real work is performed. The file server includes an embedded ROM file system.
- The Filesystems component provides file-system plug-in implementations of LFFS and FAT file systems. FAT is the native format for externally visible drives, for example those implemented on removable media.
- The FAT Filename Conversion Plug-ins support filename conversion from and to Unicode.

User-Side Hardware Abstraction Collection

This API provides Get and Set functions to query and set information about specific hardware features from the user-side, providing a way to access and control many device-specific features independently of the hardware platform. See Figure 10.4.

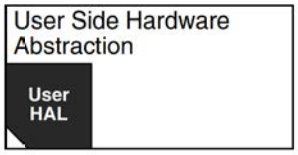


Figure 10.4 User-Side Hardware Abstraction components

Table 10.2 User-Side Hardware Abstraction Components

Component Name	Development Name
User HAL	HAL_EKA2

This component is deprecated for application use. The intended users are system components running on the user-side and needing to access hardware properties, for example fault and exception, memory-page size, timer-tick period, screen properties (whether a screen backlight is present or not, setting the display contrast), and so on.

Text-Mode Shell Collection

Together, the Text Shell and the Text Window Server that supports it make the base layers of the operating system independent of higher level services (for example graphics and windowing support as well as GUI-based application support), allowing functional builds of the base to support porting and other low-level development. See Figure 10.5.

Table 10.3 Text Mode Shell Components

Component Name	Development Name
Text Window Server	EWSRV
Text Shell	ESHELL

- The Text-Window Server supports the Text Shell, using a text-mode display driver to provide standard VGA/LCD screen displays on local hardware as well as VT100 terminal emulation over a serial line.
- The Text Shell provides a console-like (command-line) interface to basic operating system services, for example navigating the file system and launching executables, for use in porting, testing, and low-level development in which only the base layers of the system are built.

Low-Level Libraries and Frameworks Collection

This collection contains a number of basic system frameworks and libraries which are used throughout the system as well as by applications.

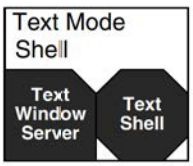


Figure 10.5 Text-Mode Shell components

It includes the Plug-in Framework, which provides a uniform and secure plug-in definition and loading mechanism, Store, which implements the Symbian OS persistence model, and a varied collection of system utilities. These include a cryptography library, which implements both weak and strong versions of standard cryptography algorithms, a Zip compression library, and a basic application utilities library (BAFL).

Table 10.4 Low-Level Libraries and Frameworks

Component Name	Development Name
Cryptography Library	`CRYPTOGRAPHY`
Zip Compression Library	`EZLIB`
Plug-in Framework	`ECOM_ONGOING`
Power and Shutdown Management	`DOMAIN`
Application Utilities	`BAFL`
Feature Registry	`FEATREG`

Among the more recent components (new in Symbian OS v8) is the Central Repository which is provided to store state and settings information that need to be persistent for clients, for example default filenames, locale settings, user preferences, etc. See Figure 10.6.

- The Cryptography Library implements (since Symbian OS v7) non-RSA-based cryptographic support for symmetric and asymmetric ciphers, hash functions, random number generation, and password encryption.
- The Zip Compression Library is a port of the zlib compression library (see relevant RFCs e.g. RFC1950) used to support compression and decompression of SIS files (the native Symbian OS installable application format) and Java Archive (JAR) files, and for PNG decompression.

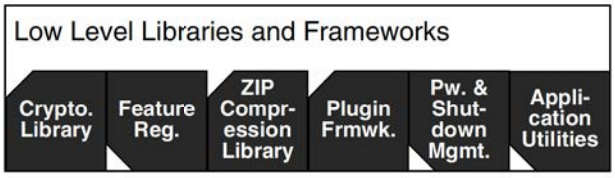


Figure 10.6 Low-level libraries and frameworks

- The Plug-in Framework is a framework and server for plug-in interface implementations. It defines the standard base classes used by conforming plug-ins and a client-side API used by framework clients to locate and load plug-ins on demand. It manages a registry of available plug-ins and implements security policy mechanisms (e.g. capability policing).
- The Power, Memory and Disk Management component is a customizable user-side power manager supporting policy-driven power management via power domain 'profiles' at device switch-on and switch-off. It includes a notification service (the so-called 'Shutdown Server') to clients to provide 'save data' and 'release resources' notifications in case of switch-off, low memory and similar events.
- The Application Utilities component, known to developers as BAFL, provides an assortment of utilities organized as a single library DLL including utility classes for resource-file handling and file finding, and implementations of string pools and descriptor arrays.
- The Feature Registry (introduced in Symbian OS v9.2) provides an API enabling run-time queries to discover whether known but optional features are supported on the run-time platform.

Character Conversion Collection

This collection provides a character-code conversion framework and plug-ins. See Figure 10.7.

Table 10.5 Character Conversion Components

Component Name	Development Name
Character Encoding and Conversion Framework	`CHARCONV_ONGOING`
Character Encoding and Conversion Plug-ins	`CHARCONV`

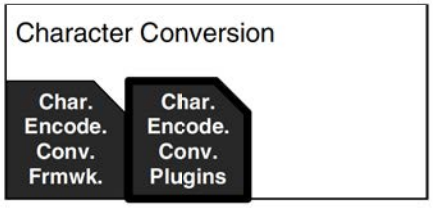


Figure 10.7 Character Conversion components

- The Character Encoding and Conversion Framework supports conversion of text between Unicode and non-Unicode character sets. Symbian OS native text formats are Unicode.
- The Character Encoding and Conversion Plug-ins provide conversion between a variety of ASCII and UTF-7 and UTF-8 text formats. The Unicode text format is UTF-8.

Persistent Data Storage Collection

The persistence model, plus the DBMS abstraction implemented as a layer around it, provides an SQL-interface for database applications. It also includes the Central Repository that provides a uniform approach to persistent settings management. See Figure 10.8.

Table 10.6 Persistent Data Storage Components

Component Name	Development Name
Store	`STORE`
DBMS	`DBMS`
Central Repository	`CENTRALREPOSITORY`

- The Store component defines the Symbian OS persistence model based on the two abstractions of streams and stores, providing an application data-storage model which shields applications from the underlying File Server implementation.
- The DBMS component defines a general relational database access API and implementations for fast client-side-only exclusive access and slower client–server-based shared-access databases. Databases can be manipulated either through a native API or a subset of SQL.
- The Central Repository component provides a single persistent store for global settings as well as a notification mechanism allowing clients to register interest when settings change. The Central Repository was introduced in Symbian OS v8.

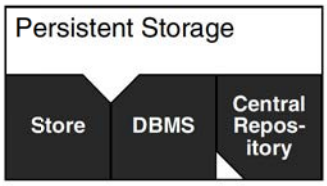


Figure 10.8 Persistent Data Storage components

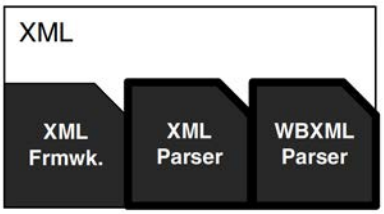


Figure 10.9 XML components

Table 10.7 XML Components

Component Name	Development Name
XML Framework	`XML`
XML Parser	`XMLPARSERPLUGIN`
WBXML Parser	`WBXMLPARSER`

XML Collection

XML support includes an extensible framework and parser plug-ins for parsing and validating XML documents (see Figure 10.9).

- The XML Framework provides an extensible framework for XML parsing based on a parser model similar to SAX-2.0 and supporting DTD and processing plug-ins (for example, validators and auto correctors) as well as parser plug-ins.
- The XML Parser component is a non-validating parser plug-in for XML 1.0.
- The WBXML Parser component is a parser plug-in for WAP Binary XML (WBXML).

Media Device Framework Collection

The Media Device Framework (see Figure 10.10) defines standard hardware acceleration APIs which are used by the Multimedia Framework and its clients, enabling multimedia controller plug-ins to communicate with hardware accelerator adaptors through standard interfaces.

- The Media Device Framework contains standard acceleration APIs for audio, video, MIDI, and Automatic Speech Recognition (ASR).
- Media Device Framework Plug-ins is an ASR Client Utility API that provides speaker-independent speech recognition to the Multimedia

Framework and directly to other clients wanting to interface to speech-recognition hardware (or software emulations).

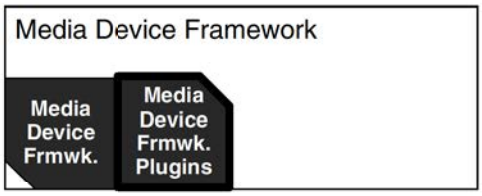


Figure 10.10 Media Device Framework

Table 10.8 Media Device Framework Components

Component Name	Development Name
Media Device Framework	MDF
Media Device Framework Plug-ins	AUDIODEVICE, MDFAUDIOHWDEVICEADAPTER, VORBISDECODERPROCESSINGUNIT, VORBISENCODERPROCESSINGUNIT, MDFVIDEODECODEHWDEVICEADAPTER, MDFVIDEOENCODEHWDEVICEADAPTER

11

The Kernel Services and Hardware Interface Layer

11.1 Introduction

The Kernel Services and Hardware Interface layer (see Figure 11.1) is the lowest layer of Symbian OS. It contains the Symbian OS kernel and supporting components.

These include the kernel-level components which must be customized in order to bring up a minimal build of the operating system on new hardware (although a typical port entails customizing other components too).

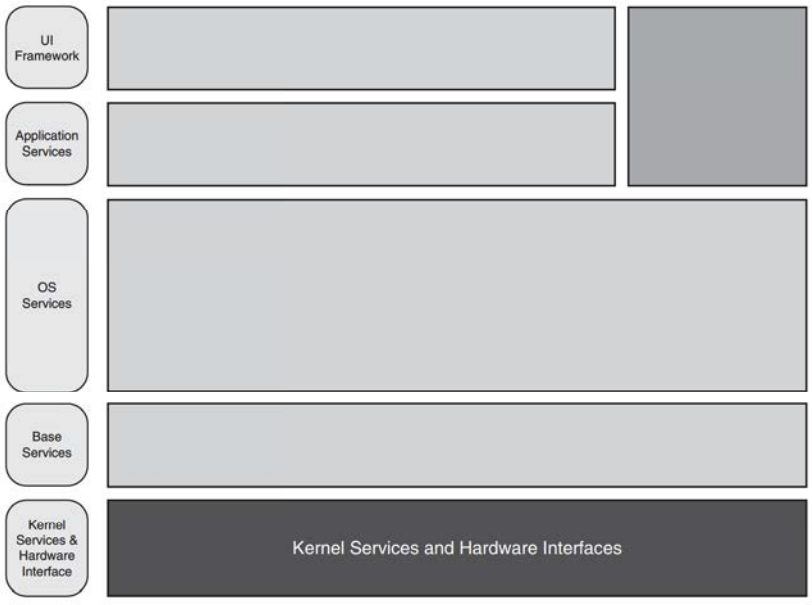


Figure 11.1 Kernel Services and Hardware Interface layer in the system model

The layer boundary also marks the 'kernel side' boundary; all components which run in privileged mode in the runtime system are included within the layer.

11.2 Purpose

The Kernel Services and Hardware Interface layer is the foundational layer of Symbian OS. It includes the kernel and all the supporting infrastructure needed to boot and run the kernel on the underlying hardware platform. It is responsible for fundamental operating system services:

- bootstrapping the physical or emulated device to provide the basic initialization of the hardware
- creating and managing the fundamental operating system kernel abstractions, for example, threads, processes, memory address spaces, and other resources including timers, mutexes, and so on
- scheduling, pre-emption and interrupt handling
- access to devices, providing the device-driver framework and device drivers that abstract device hardware and implement the two-tier logical and physical device driver model
- encapsulating the kernel–user boundary; all processes which run in privileged mode originate from this layer
- encapsulating the lowest level of an operating system port ('base port') to new hardware
- insulating all higher layers from actual hardware.

The system model collects the kernel and kernel extensions, device drivers, and the other hardware abstraction components which are required for hardware porting, into a single Kernel Architecture block. (Versions of the system model for Symbian OS v8 have two Kernel Architecture blocks, for each of the EKA1 and EKA2 kernel versions.)

Two small collections sit within the layer but outside the block. These collections each have a single component which is independent of the kernel version but which requires customization in a new port. These components implement locale support, which is used by the kernel, and the screen driver.

From Symbian OS v9, the Kernel Architecture block is organized to reflect the basic architecture of the kernel side of the system, as well as the recommended structure of a base port. The kernel includes extensions that implement the device-driver framework, providing a two-layer logical–physical device-driver model in which logical device drivers abstract a generic device interface and physical device drivers drive the actual hardware. Below the kernel, abstraction of device hardware

is divided between the Application-Specific Standard Part (ASSP), an off-the-shelf integrated CPU, and the Variant components. The ASSP component contains ASSP-specific code that is otherwise hardware-agnostic (it supports the specific silicon package used in a product, typically a standard part containing the CPU core and custom chips). The Variant components contain hardware-dependent code which is specific to a product, for example hardware-specific flash-memory translation.

11.3 Design Goals

Releases up to and including Symbian OS v8 shipped with the original kernel, EPOC Kernel Architecture 1 (EKA1). Symbian OS v9 and later releases ship with the new kernel architecture of the EKA2 'real-time' kernel.

At the highest level, the design goals of the kernel layer of Symbian OS are common to both kernel versions:

- provide an operating system kernel optimized for its device class – palmtop and smaller
- optimize for ROM-based execution – XIP- or RAM-shadowed execution
- optimize for mobile – no fixed wires
- optimize for battery operation – anything from the two 'AA' batteries of the original Psion Series 5 to the latest mobile phone rechargeable battery
- target consumer-oriented devices – for 'ordinary' non-technical users.

Immediate performance goals follow:

- meet the requirements of the device class – in terms of the operating system image size, start-up time, task-loading and task-switching times, its ability to run forever, and overall robustness
- meet consumer-device goals – robustness in the face of typical failure scenarios, for example out-of-memory, no signal, low battery or sudden battery removal, media card removed in mid-write, disk full but camcorder still running, and so on
- provide a highly portable operating system kernel – to enable porting to multiple hardware architectures in as pain-free a way as possible
- support typical licensee product models, that is, the product line or product family principle – multiple minor hardware revisions follow from an initial 'lead product' and porting effort should scale down significantly between a first port and subsequent incremental ports.

Compared with the original kernel, EKA2 is explicitly designed to make porting easier by improving the modularity of the kernel and the

structuring (and packaging) of its supporting components. Thus the core kernel is independent of both ASSP and variant. In contrast, in EKA1 the separation between hardware-dependent and hardware-independent code was less clear-cut, and hardware support was less cleanly partitioned between the ASSP and variant.

It is also important to remember Symbian's origins as an application-centric operating system, which determines additional design goals:

- provide a fully programmable platform – enabling user-installable applications as well as a complete native application set
- provide a fully graphical system which is intuitive to use – with full interactive GUI, multitasking and instant task switching.

From the beginning, Symbian OS has also been strongly focused on international markets, with early support for non-Western scripts (for languages such as Chinese, Arabic, Thai and Hindi):

- Unicode multi-byte characters supported throughout the system
- easy localization
- non-Roman and multidirectional script display.

Increasingly, the application emphasis has evolved from PIM applications (calendars, contacts books and so on) toward high-data bandwidth applications, including camcorder applications and mobile digital TV, following the trend of increasingly multimedia capable devices.

As well as requiring the architecture to support ever higher data rates, this overall shift in the market away from PDA-style products towards mobile phones has led to an important evolutionary goal and, in particular, to specific requirements on the EKA2 kernel. The kernel was to be capable of supporting typical licensee phone hardware architectures, including one-core and dual-core variants, and Symbian-only as well as 'partner operating system' configurations (requiring cooperation with a real-time 'partner' operating system driving the baseband hardware and software).

Arguably the most critical design goal follows from the above: provide a highly adaptable and evolvable kernel architecture capable of change in a rapidly evolving technology, product and market context.

The strength of the kernel architecture is demonstrated by its stability and continued fitness for purpose in the face of rapid change – for example, the almost complete transformation of the mobile phone in less than a decade, from the pre-Symbian OS basic phone of the mid-1990s to the PDA–phone 'smartphone' hybrids with which Symbian OS entered the phone market to today's full multimedia devices.

11.4 Overview

Symbian OS has a microkernel architecture,[1] which means that the responsibilities of the kernel are kept to an essential minimum. The design approach is to implement a minimal set of operating-system primitives in the kernel, on which higher-level, generic operating system services can be built, the goal being to keep the kernel small, and therefore fast, and to keep its complexity low, to achieve high reliability and predictability.

Simplistically, kernel responsibilities are divided between implementing suitable primitives for use by the higher layers of the operating system and interfacing to the underlying hardware platform. Surrounding the kernel itself are the additional components required to provide complete hardware support.

Because the kernel layer is the interface to the hardware platform, it is dependent on the hardware. To port the operating system to new hardware entails porting the kernel layer. An important design consideration, therefore, is to optimize ease of porting by isolating hardware dependencies. The design of the kernel and its supporting components is highly modular, to make porting simpler.

An important distinguishing feature of Symbian OS is its optimization for ROM-based systems. Symbian OS was designed to be built into device ROM and executed in place without requiring loading into RAM, in contrast to more conventional systems (including Linux/Unix and Microsoft Windows), which are designed to be loaded from the file system into RAM before executing.

Supporting ROM-based systems has become more complex as memory technologies and hardware architectures have evolved to keep pace with the burgeoning requirements for storage capacity. The latest releases of Symbian OS are optimized for multiple hardware architectures and memory types, including the latest NAND-flash-based systems as well as more conventional NOR flash. On NOR-flash systems, Symbian OS is executed in place (XIP). On NAND flash, which is not byte-addressable, Symbian OS shadows itself to RAM from where it executes. In both cases, it provides a translation layer to interface the filing system to the flash drive.

11.5 EKA1 and EKA2

The origins of EKA1 go right back to the first releases of the operating system. The original architecture of the Symbian OS kernel was driven by the need to provide a robust platform for a PDA-centric (and, therefore, application-centric) operating system. Almost from its first release, however, Symbian OS has been evolving to meet the high data-throughput

[1] There are some aspects in which it is more hybrid than pure (see the detailed discussion below).

and real-time requirements of more communications-centric devices (in particular, advanced mobile phones).

As early as 1998 (i.e. two years before Symbian OS v6 was released) a 'real-time' project began in the kernel team to investigate the issues involved in providing real-time support and to prototype a solution.

The eventual result of that work was a new, real-time-capable kernel, EKA2 (also known as EpocRT), benchmarked in terms of its ability to directly support a full mobile phone signaling stack. It was intended for release in Symbian OS v7 and reached the market in Symbian OS v8, becoming the standard kernel in Symbian OS v9. (In Symbian OS v8.1, customers were offered a choice between the EKA1 and EKA2 kernels.) Even so, the new kernel's initial selling point for customers was probably less its real-time capabilities than its support for the new Platform Security architecture, which had become commercially necessary. Platform Security requires kernel support to police security policies as part of its inter-process communication (IPC) mechanism. While Platform Security was introduced in stepped phases to be compatible with the original kernel, the full features of Platform Security are only available in a system running EKA2.

EKA2 was designed to be closely compatible with EKA1. In important respects, the two are functionally equivalent, as evidenced by the choice of using either EKA1 or EKA2 in Symbian OS v8.1. The critical difference is that EKA2 is designed to offer true real-time behavior.

11.6 Singleton Component Collections

The Kernel Services and Hardware Interface layer consists of the Kernel Architecture block (or blocks, in the case of releases that include both kernel versions) and two singleton component collections (see Figure 11.2) containing components that, while they are not part of the kernel architecture proper, nonetheless can be counted as belonging on the kernel side of the kernel–user boundary.

Localization Collection

This component is a customizable plug-in that implements locale-specific settings including standard strings (for example, day and month names),

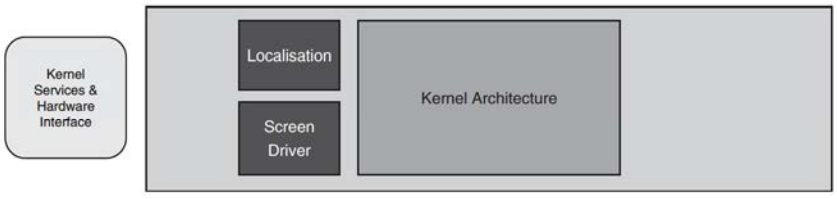


Figure 11.2 Localization and Screen Driver collections



Figure 11.3 Localization components

distance units, currency symbols, date and time formats, collation orders, and so on. Standard locales, including Japanese and several Chinese variants, are provided with the system.

Locale Support is included in the Kernel Services layer because it implements various strings used directly by the kernel (e.g. default system messages). It is loaded by the User Library.

Table 11.1 Localization Components

Component Name	Development Name
Locale Support	`LOCE32_ONGOING, ELOCL`

Screen Driver Collection

This component implements the generic operations defined by the Bit GDI to manipulate the physical memory map of the device display or bitmap memory map. (Typically, in-memory bitmaps and the display memory map are addressed in the same way in hardware, hence a common interface is provided to both.) It supports dual screens, which feature in flip-phone designs. The Screen Driver forms part of a base port to new hardware.

Table 11.2 Screen Driver Components

Component Name	Development Name
Screen Driver	`SCREENDRIVER`

11.7 Kernel Architecture Block

In one sense, the Symbian OS kernel has always been larger than a microkernel, since in both EKA1 and EKA2 it includes extensions and

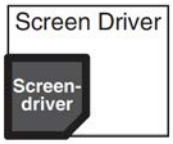


Figure 11.4 Screen Driver components

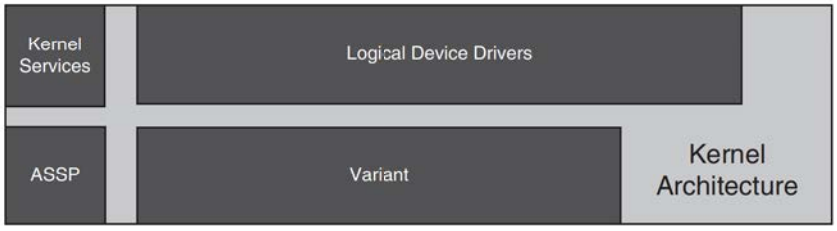


Figure 11.5 Kernel Architecture block in Symbian OS v9

device drivers. In another sense, in EKA2 (see Figure 11.5) it is even smaller, with a true nanokernel at its core.

However, both kernel architectures have true microkernel properties. For example, major services such as the File Server and the User Library, as well as all graphics and communications services, including networking and telephony, remain outside the kernel and are run as user-side processes. This is in contrast for example to the monolithic kernel architectures of both Linux and Microsoft Windows.

A microkernel limits the kernel responsibilities to a small set of core functions, and builds higher-level operating-system functions on top of a small set of kernel primitives. The microkernel can thus be kept small and fast. Another important feature of microkernel architecture is that kernel functionality is deliberately simplified; more complex higher-level behavior is moved out of the kernel onto the user side. The principles parallel those of Reduced Instruction Set Computer (RISC) versus Complex Instruction Set Computer (CISC) processor design, with broadly comparable arguments in favor.

From microkernel architectures, the Symbian OS kernel borrows the following features:

- a message-passing framework for the benefit of user-side servers
- networking and telephony stacks as user-side servers
- file systems implemented as user-side servers.

At the same time, for performance reasons Symbian OS compromises on microkernel purity by allowing kernel extensions and including the device-driver framework in the kernel. However, device drivers are not embedded in the kernel binary but follow the typical Symbian OS design pattern of being implemented as run-time loadable and unloadable plug-ins.

From monolithic kernel architectures, the Symbian OS kernel borrows the following features:

- kernel-side device drivers
- scheduling policy implemented in the kernel.

The test case for the success of the EKA2 kernel architecture is its ability to support the real-time requirements of a GSM/wCDMA or CDMA phone-signaling stack ([Sales 2005, p. 778]). To do so requires that real-time guarantees can be given for key services, most importantly interrupt latencies, thread latencies and context switches. ('Real-time' in this context means deterministic and bounded by a predictable and known time; which is not quite the full definition.[2])

The rationale for providing real-time support is two-fold. First, as phones become more complex and add more custom hardware, particularly to support multimedia functions, interrupt latencies become increasingly critical to data throughput. Secondly, there is a specific goal to enable the Symbian OS nanokernel to operate as a true real-time operating system capable of hosting the baseband software. The baseband (phone software stack or 'modem') in a mobile phone requires real-time support in order to respond to the timing requirements of the signaling stack.

Typical phone designs host the baseband stack on a real-time operating system. In a phone that also provides sophisticated application-side software including, for example, a full GUI, a second, application-centric operating system is dedicated to providing the application support. The 'dual operating system' design most commonly also implies a 'dual core' two-processor design, in which dedicated baseband and application-side processors host the respective operating systems and, in many cases, also own dedicated peripheral hardware including memory. Adding real-time capability to Symbian OS is intended to enable 'single operating system', and hence 'single core', designs.

The EKA2 architecture is based on a nanokernel which is designed to have sufficient functionality and the real-time properties required to directly host a GSM, wCDMA or CDMA phone stack. Phone stacks are not yet commodity items and most have been written to interface to an existing real-time operating system (RTOS), whether bespoke or off the shelf. The EKA2 nanokernel therefore supports a 'personality' layer mechanism that enables an interface layer to be written between a given RTOS and the software above it, while mapping calls into the interface to the underlying functionality of the nanokernel. This provides a way of running a baseband stack directly on Symbian OS without having to first port the stack. Writing a personality layer is a small task compared to that of porting an existing phone stack or writing one from scratch to the Symbian OS nanokernel interface.[3]

The kernel re-architecture had a secondary goal of improving the modularity of the kernel. The EKA1 architecture includes an undesirable

[2] See [Sales 2005, Chapter 17] for a detailed discussion of what real-time means and how Symbian OS meets real-time requirements.

[3] The task of creating a new phone stack, from design to full type approval, can take 10s to 100s of man-years of coding effort.

degree of hardware dependency. Although at the lowest level of the kernel an EKA1 'variant' DLL encapsulates many of the device-specific hardware dependencies, many ASSP-specific assumptions are contained in generic EKA1 kernel code, meaning that customization of kernel code is still required to move to a new ASSP architecture (or, at the very least, the kernel needs to be recompiled).

In the EKA2 architecture, all peripheral-related code (i.e. code which is ASSP-specific, but not specific to a particular licensee product) moved out of the kernel into a separate ASSP DLL. This provided better isolation of the kernel from the porting effort and enabled a more flexible approach to porting (since a licensee can choose how to structure the port between the Variant and ASSP DLLs, to better support families of similar but not identical devices; indeed, a licensee can even choose to dispense with the ASSP DLL for a one-off port).

Architecture

The project that led to the creation of EKA2, the 'real-time' kernel, had a number of goals:

- to enable the creation of single-core, single-operating-system products in which baseband software, for example a GSM protocol stack, executed on the same processor as the application software, supported by the same operating system
- to improve average overall performance, as well as portability and robustness
- better timer resolution, easier debugging, a better emulator (i.e. a more faithful 'virtual' port to Microsoft Windows), and general architectural housekeeping.

EKA2 meets all of those goals. In particular, it is highly portable, running on X86 as well as many flavors of ARM processor architectures (ARM720/920/SA1/Xscale); on systems with different Memory Management Unit (MMU) styles, including no MMU;[4] and on multiple ASSPs. Its architecture (see Figure 11.6) is highly modular and carefully layered to isolate hardware dependencies.

At the heart of the design, the nanokernel implements essential operating primitives and supports real-time guarantees for interrupt latencies, thread latencies and context-switching time bounds. The nanokernel is responsible for the most basic thread scheduling, synchronization and

[4] Realistically, the 'no MMU' option is intended as an aid to porting rather than a supported target architecture. The security model, for example, depends on an MMU being present to enforce memory protection between processes.

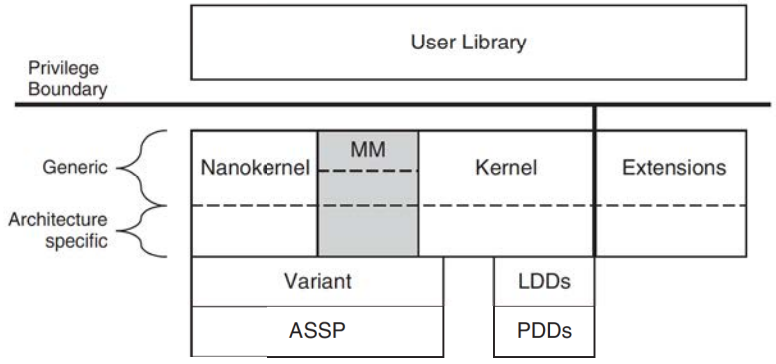


Figure 11.6 Kernel architecture for EKA2

timing functions. Extending the nanokernel, the kernel proper provides higher-level operating-system services compatible with the EKA1 kernel.

Both the nanokernel and the kernel are isolated from hardware dependencies by the Memory Model, ASSP, and Variant modules. The Memory Model provides per-process address spaces and inter-process data transfer. The Variant represents the specific, 'off-chip' system hardware, while the ASSP represents the core silicon package.

The device-driver model and extension mechanism control peripheral devices and provide client interfaces. (Extensions are statically linked device drivers.)

Kernel Responsibilities

The Symbian OS Kernel implements the operating-system primitives on top of which generic services are built by higher-level components (such as the File Server and the User Library). In particular, the kernel, including its extension mechanisms, implements:

- the thread and process models that provide the underlying basis for all code execution, including process creation and termination, code loading, thread scheduling and the scheduling policy
- memory management, including Direct Memory Access (DMA), which is an essential service underlying the process model
- process protection and IPC mechanisms that guarantee process independence while allowing processes to cooperate; IPC is at the heart of the client–server model and policing of the platform security model
- the device-driver model, which provides the device-level interface for system clients and applications
- interrupt management, which is exclusively the responsibility of the kernel

- the power model, which provides an interface for higher-level clients to manage and respond to the device power state
- logical device drivers (LDD), which are implemented as plug-ins to the device driver framework and provide a high-level device interface (i.e. LDDs support classes of device, e.g. Ethernet ports)
- physical device drivers (PDD), which are implemented as plug-ins to LDDs and provide the low-level interface to actual hardware present on a device (for example, a specific Ethernet card)
- various other high-level drivers (for example, accelerator plug-ins to the Media Device Framework), which are not implemented as LDDs, operate at an equivalent level of abstraction.

Kernel Executive Calls

Executive calls are the mechanism used to call into the kernel from user-side programs. While the interface is defined by the User Library, the underlying mechanism is a kernel-side software interrupt dispatch table that decodes software interrupts generated by invocation of the CPU software interrupt instruction into a specific executive call. From the user side, the mechanism is entirely wrapped by methods of the User Library static classes.

Inter-Process Communication

IPC is supported by an asynchronous message-passing mechanism, based on Executive calls, which is the basis for client–server communications, as well as inter-thread communications used in system-level programming (for example, device-driver programming). EKA2 also introduces IPC based on message queues, which is distinct from client–server IPC, and enables shared chunks.

Publish and Subscribe

EKA2 introduces Publish and Subscribe, a mechanism for defining global properties whose values may then be 'published', that is, updated by the property owner, to 'subscribers' who have a dependency on the value. Publish and Subscribe is, in effect, a system-wide asynchronous notification mechanism to which interested clients can subscribe to track the changing values of arbitrary properties. Broadly speaking, it is an asynchronous IPC mechanism although, more precisely, it is really an inter-thread mechanism, since both publishers and subscribers are threads.

Properties are single data values, that is 32-bit integers or 'string' values (strictly speaking, descriptors that contain byte or text data, including

Unicode text) of up to 512 bytes in size. Larger property arrays of up to 64 KB can also be defined but do not have the same deterministic operation.

Real-Time Processing

In EKA1, the kernel is single-threaded. In EKA2, the kernel is multithreaded and all threads are pre-emptible. Interrupt latencies and process switching are time-bounded (whereas they are potentially unbounded in EKA1).

Memory Model

In EKA2, all MMU-related code is moved into a separate module (the 'memory model'), which is linked against the kernel at build time. (EKA2 also uses a different memory-map base address.) The kernel itself is thus MMU-agnostic. The following memory models are supported:

- the moving model (ARMv4 and ARMv5 architecture) is functionally equivalent to EKA1; it uses a single memory-page directory within which entries are moved when address spaces are switched
- the multiple model (ARMv6) uses per-process memory-page directories
- the single model has no address-space paging but uses a single address space (it is used with CPUs without an MMU or to simplify the early stages of porting)
- the emulator model for memory management on a PC.

Device-Driver Model

In EKA2, the device-driver model is made more flexible to allow multiple user-side client device-driver requests to be handled by a single kernel thread, which in effect serializes access and therefore simplifies device-driver programming.

Device drivers are DLLs that allow code running in Symbian OS to communicate with hardware in the variant or kernel extensions. Device-driver DLLs are loaded into the kernel process by explicit load commands from the user side. (In contrast, kernel extensions are automatically loaded during the kernel boot process.)

User-side code accesses a device driver through a specific API provided by the kernel, which provides functions to open a channel to a device driver and to make requests. These functions are protected and the device-driver author provides a derived class to implement functions that are specific to the device driver.

Device-driver DLLs come in two types – logical (LDD) and physical (PDD) device drivers. Logical device drivers provide an abstracted representation of hardware and typically support functionality common to a class of hardware devices. Physical device drivers support specific devices. Thus, for example, there is a single serial communications LDD (ECOMM) that supports all UARTs, providing buffering and flow-control functions. On a particular hardware platform, different physical UARTS (for example, RS232 and infrared) are supported by device-specific PDDs.

Porting

Modularity is improved in EKA2 and more flexible porting strategies are supported. Changes to the ROM building tools also enable greater ROM build-time (rather than compile-time) customization.

Emulator

The EKA2 Emulator is written as a true port of Symbian OS to a (virtual) hardware platform and, in that respect, is like any other variant port. This is unlike the EKA1 emulator implementation, which mapped native Symbian OS system services to their best equivalents on the Microsoft Windows host (i.e. trying to make the Symbian OS kernel API work on Microsoft Windows, so that the underlying services, including the scheduler, are all Microsoft Windows services).

As a result, the EKA2 emulator is a much more faithful representation of Symbian OS although, since Microsoft Windows forces the emulator executable to be a single process, the emulator must use Microsoft Windows threads to emulate Symbian OS processes. A deliberate goal of these changes is to make it easier to implement an emulator on platforms other than Microsoft Windows.

Power Management

EKA2 introduces a new power management framework, which is intended to improve flexibility by supporting a wider range of hardware and by separating policy from mechanism. The new framework is based on the concept of power domains. (A domain is a set of processes that share the same power management characteristics.)

A user-side domain server provides a single point of interaction with the kernel. Policy (the definition of power states) is implemented in a customizable DLL.

On the kernel side, a power manager is embedded in the core kernel, which implements the power-management executive calls. An ASSP-specific power controller is implemented in a kernel extension and manages the different power states and sleep modes supported by the ASSP.

SDIO Support

The MMC/SD bus controller is extended in EKA2 to support SDIO cards (SD cards that provide an interface to a hardware device as well as memory). For example, an SDIO card could provide access to a camera and to some memory.

Unicode Support

Unicode strings are not directly supported within the EKA2 kernel, and therefore all kernel objects (processes, threads, and so on) have ASCII names, implying that user-side code should use only the ASCII subset of Unicode when creating such objects.

User Library

In EKA2, the User Library is available only on the user side and a kernel-specific utility library is used on the kernel side. In EKA1, both user-side and kernel-side code were linked against the User Library DLL.

Summary of Major Kernel Differences

- EKA2 offers real-time support.
- EKA2 supports alternative memory models.
- Device-driver implementation in EKA2 is simplified by supporting an alternative, serialization approach to handling multiple user-side requests in a single kernel thread (in the case where multiple device drivers share a kernel thread, which is optional).
- EKA2 includes improved modularity and greater flexibility for porting, a new power-management model, emulator improvements, and support for SDIO cards.
- EKA2 does not support Unicode strings inside the kernel and the User Library is available only on the user side. (The kernel implements its own User Library subset.)

Overall, there are some small system-wide impacts from these changes:

- EKA2 threads use more RAM than EKA1 threads (4 KB to support a per-thread kernel-mode stack)
- In EKA2, memory chunks are limited to 16 per process (for the Moving Memory model, in order to support deterministic operation although context-switching times remain nondeterministic). For the Multiple

Memory model, there is no chunk limit and, in addition, it provides fast, deterministic context switching.

Hardware Interface, Base Porting and Reference Hardware

Kernel extensions provide the interface to the platform hardware (the device-driver model provides the interface to specific devices). The modular, extension-based architecture is designed to allow for flexible customization when moving the operating system to new hardware. In particular, it is designed to enable generic platform dependencies encapsulated by the Application-Specific Integrated Circuit (ASIC) or ASSP, for example, processor type, MMU architecture and standard peripherals such as DSPs and LCD drivers, to be isolated from the device-specific hardware such as the flash-memory interface.

Platform dependencies are typically encapsulated as ASSP dependencies and device-specific dependencies as 'Variant' dependencies.

- An ASSP extension module implements hardware-dependent support for a given ASSP.
- A device-specific 'variant' extension module implements other device-specific hardware support, for example, for peripherals that are not standard to the ASSP.
- Other standard extensions include the power framework and the peripheral bus and USB controllers.

To provide a reference point for porting work, each release of Symbian OS is built and warranted against the hardware reference platform. The hardware reference platform for Symbian OS v8 releases was the Intel Lubbock development board. For Symbian OS v9, the hardware reference platform is based on the Texas Instruments OMAP development boards.

For Symbian OS v9.0 and v9.1, the hardware reference platform is the H2 development board with OMAP 1623 (ARMV5-based core):

- other versions of the H2 boards are not officially supported
- includes a DSP, SD/MMC/SDIO card, USB, camera, display (240×320, rotatable, 8 or 16 bits per pixel), NAND flash
- operating system installation from MMC or serial loads either into RAM or into NOR Flash on the board
- JTAG (IEEE 1149.1) is also supported.

From Symbian OS v9.2, the hardware reference platform is the H4 development board with OMAP 2420 (ARMv6-based core):

- adds USB for bootable image source to H2
- introduces some NAND-flash differences compared to the H2 board.

11.8 Kernel Architecture Component Collections

The Kernel Architecture block in the system model contains four separate collections (see Figure 11.7): Kernel Services, Logical Device Drivers, ASSP, and Variant.

Kernel Services Collection

The EKA2 component consists of the nanokernel, which is the real-time kernel core, and the operating system kernel that builds the basic threading, process and memory models on top of it.

Table 11.3 Kernel Services Components

Component Name	Development Name
Kernel Architecture 2	`E32_EKA2`

Logical Device Drivers Collection

Logical (see Figure 11.9) device drivers (LDDs) are plug-ins to the kernel device-driver framework that provide the logical abstraction of hardware

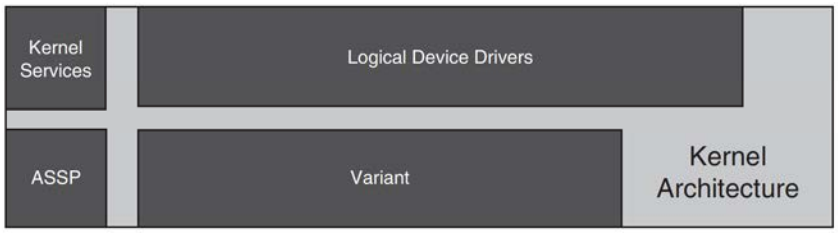


Figure 11.7 Kernel Architecture collections



Figure 11.8 Kernel Services components

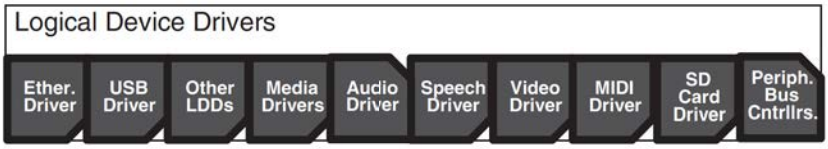


Figure 11.9

Table 11.4 Logical Device Drivers

Component Name	Development Name
Ethernet Driver	ETHERDRV
USB Driver	USBC
Audio Driver	SOUNDDEV
MIDI Driver	DEVMIDI
Speech Driver	DEVASR
Video Driver	DEVVIDEO
Other LDDs	
Media Drivers	MEDUSII, MEDUSII_CRASHLOG, MEDUSIIS
SD Card Driver	SDCARD4C
Peripheral Bus Controllers	EPBUS

devices, and accept the physical device driver (PDD) plug-ins, which communicate with real hardware.

Symbian OS supplies specific Ethernet and USB drivers, as well as hardware accelerator plug-ins used by the Media Device Framework, which form part of the hardware abstraction for multimedia devices.

- The Ethernet Driver is a logical device-driver implementation for Ethernet cards, including the emulator.
- The USB Driver is a logical device driver for USB. The standard USB software architecture on Symbian OS supports dynamically configurable USB 2.0 device functionality.
- The Audio, MIDI, Speech and Video accelerator API plug-ins to the Multimedia Device Framework (MDF), the lowest-level framework supporting multimedia services, are used by MDF controllers. They all include hardware- or kernel-dependent components.

- DevVideo is the hardware-abstraction layer for video decoding and encoding acceleration enabling playing and recording of video; it includes a client API that enables policy management (e.g. request contention and file-type matching).
- DevMIDI is the API that supports hardware-accelerated MIDI engines.
- DevSound is the hardware-abstraction layer for digital audio acceleration enabling Playing, Recording, Conversion and Tone generation of sounds; it includes a client API that enables policy management (e.g. request contention and file-type matching).
- DevASR is the hardware-acceleration API for Automatic Speech Recognition, allowing the computationally intensive speech-recognition algorithms to be performed in hardware, where present.

- The peripheral bus controllers for supported variants are implemented as kernel-side DLLs that interface media and I/O device drivers to PC-card or MMC-card socket hardware.

ASSP Collection

Hardware dependencies divide between ASSP dependencies, based on properties of the CPU core and the peripherals packaged with it in the same silicon chip, and additional 'off-chip' hardware peripherals.

Table 11.5 ASSP Components

Component Name	Development Name
ASSP	OMAP 1623

Isolating the ASSP-dependent code allows it to be reused on multiple systems that use the same ASSP. OMAP 1623 is the ASSP supported in the Symbian OS v9 hardware reference platform, as used in the Texas Instruments H2 development board. (Other H2 ASSPs are not supported.) The ASSP module contains source code tailored to a range of different microprocessors (e.g. ARM720/920/SA1100/Xscale). See Figures 11.9 and 11.10.



Figure 11.10 ASSP components

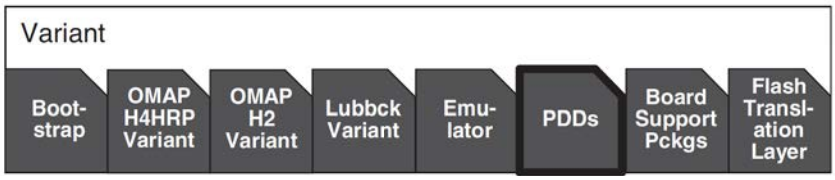


Figure 11.11 Variant components

Table 11.6 Variant Components

Component Name	Development Name
Bootstrap	BOOTSTRAP
Emulator	WINS_VARIANT_EKA2
Lubbock Variant	LUBBOCK_EKA2
OMAP H2	OMAP_H2
OMAP H4	OMAPH4HRP
PDDs	
Board Support Packages	
Flash Translation Layer	UNISTORE2_DRIVERS

Variant Collection

The Variant collection contains components which are associated with off-chip hardware, that is, which are independent of the ASSP.

- The Bootstrap component prepares hardware including memory and peripherals, maps virtual address space if an MMU is present, and starts the kernel. It includes processor-, MMU- and other device-dependent code, as well as a generic layer.
- The emulator component is treated as a hardware target variant. The Emulator runs on Microsoft Windows platforms and maps Symbian OS services and logical devices to Microsoft Windows APIs and local hardware. Single-process implementation uses Microsoft Windows threads to emulate Symbian OS processes. It is valuable for high-level programming, but the implementation creates practical issues for low-level and device-dependent programming compared to hardware targets.
- The Lubbock Variant component is Variant code for the Symbian OS v8 hardware reference platform.

- The OMAP H2 component is Variant code for the Symbian OS v9.1 hardware reference platform. (From Symbian OS v9.2 the reference hardware platform moves up to the OMAP H4 board.)
- The PDDs are physical device drivers, low-level plug-ins used by LDDs providing the device-dependent level of the two-tier device-driver model.
- The kernel requires additional loading and media translation support when running on NAND-flash devices. A special boot loader is required to move the kernel into RAM so that it can execute (since NAND flash is not byte-addressable), together with a translation layer that manages the flash device and presents a standard byte-readable and -writable interface to the file system. The Flash Translation Layer component is the original file system plug-in implementation of flash-driver support. Media drivers provide a newer implementation (implemented as conventional Symbian OS device drivers), supporting more NAND formats: small, large and OneNAND.

12

The Java ME Subsystem

12.1 Introduction

Symbian OS offers licensees an optional Java implementation. Since Symbian OS v7.0, this has been based on the Java 2 Platform, Micro Edition (Java ME) MIDP and CLDC specifications, which have become the standard for Java on mobile phones and other communicator-style devices.

Java ME is subdivided into configurations, profiles and optional packages. A Java ME configuration provides a basic[1] Java platform for a large class of constrained devices by defining a Java Runtime Execution (JRE) environment consisting of a Java language subset, a Java Virtual Machine (JVM) and a base set of necessary class libraries. It is commonly based on a subset of the J2SE APIs.

Currently, there are two Java ME configurations:

- Connected Limited Device Configuration (CLDC)
- Connected Device Configuration (CDC).

A Java ME profile sits on top of a configuration to complete the JRE by adding more high-level APIs, thereby preparing a device for a specific device category. Currently, there are two common Java ME profiles:

- Mobile Information Device Profile 1.0 (MIDP1)
- Mobile Information Device Profile 2.0 (MIDP2).

[1] See the CLDC specification at ***http://jcp.org/aboutJava/communityprocess/final/jsr139.***

Optional packages add functionality to the Java ME platform by offering standard APIs for various technologies such as advanced multimedia, 3D graphics, file system access, web services and much more.

A widely adopted standard is the combination of MIDP and CLDC to provide a complete Java platform for mobile phones and similar devices. MIDP applications are known as MIDlets. With MIDP 2.0 the specification becomes rich enough to support a wide variety of sophisticated applications.

12.2 Requirements of the Java ME Subsystem

Java is an important application environment for mobile phones, supporting a multiplatform third-party market for downloadable and installable programs (including games, utilities and media players), a standard environment for enterprise developers seeking to extend information systems to mobile phone users within organizations and a standard platform for mobile phone services, as well as a branding and personalization mechanism (through custom applications) for network operators and others in the phone retail chain.

Platform independence is an important part of Java's philosophy that applies as much to the MIDP context as to desktop Java implementations. The Symbian OS implementation of Java ME provides a standard environment for installing and running MIDlets, with access to the underlying operating system services through the supported MIDP 2.0 APIs, which include a number of optional APIs – for example, Mobile Media, 3D Graphics and File GCF – as well as the Java Technology for the Wireless Industry (JTWI) standard, which aims to standardize MIDP support for mobile phones, and the Unified Emulator Interface (UEI), a development tools standardization initiative.

12.3 Design Goals for the Java ME Subsystem

Symbian OS provides a rich and powerful host for the Java ME implementation, but also poses some great challenges. These are a result of the architectural differences between Symbian OS and the Java platform and the differences between Symbian OS components and the MIDP APIs which are built on top of the Symbian APIs.

- There are some specific mismatches between Symbian OS and Java models; in particular their threading models are incompatible.
- Symbian OS has its own native application model (for C++ applications), with which the MIDP lifecycle must be integrated if MIDlets are to have a seamless, native application lifecycle.

- Symbian OS has a rich set of native controls, localization abstractions, custom dialogs, input mechanisms, and so on, all of which are expected to be customized for look and feel by the providers of the variant user interface with which the operating system is integrated on any particular Symbian OS device. Therefore, any look-and-feel dependent aspects of the MIDP implementation must be customized for the different variant user interfaces.
- MIDlets should, as far as possible, look and feel like native applications. For Symbian OS, this poses a particular challenge since application look and feel ultimately depends on the variant user interface which is running on a given Symbian OS device.
- Symbian OS component APIs have a different design from MIDP APIs, which requires an elaborate internal architecture to bind the functionality between those two orthogonal class hierarchies.
- The Java ME subsystem as a whole can be swapped out and replaced by another. Symbian's licensees are also at liberty to pick and choose which of the JSRs they use and which they do not.

As well as overcoming these complications, the Java ME design must meet some basic design goals:

- Support near-native behavior of MIDlets to enable seamless switching between MIDlets and native applications
- Provide the fullest possible implementation (MIDP specifies both required and optional features) to ensure that the Java ME implementation on Symbian OS is highly competitive
- Enable a 'common platform' by providing a solid core system on all Symbian smartphones
- Provide customizability as an essential part of Symbian OS. Licensees may choose to vary the degree to which they support Java on a particular device by extending or limiting the default Symbian Java ME implementation, including removing or replacing it entirely.

12.4 Evolution of Java on Symbian OS

Symbian's Java support has evolved with Java itself from the earliest days of JDK 1.1.4, which had an AWT-based user interface implementation, to the PersonalJava and JavaPhone implementations of Symbian OS v6. The first MIDP 1.0 implementation on Symbian OS v7.0 (subsequently backported to Symbian OS v6.1) was followed by the arrival of MIDP 2.0 in Symbian OS v7.0s.

Through Symbian OS v8 and v9, Java delivery evolved to a rich Java ME platform supporting a significant number of optional MIDP APIs and enhanced by the latest version of Sun's CLDC 1.1 VM, which uses HotSpot technology.

Although recently the emphasis has shifted towards Symbian's licensees as the ultimate 'owners' of their own Java ME platforms, with Symbian focusing on the core of its delivery, Java has strengthened its position as an important element in any enterprise strategy, where rapid development and rollout of bespoke local applications is an important driver in the choice of devices. It is also an important enabler for rapid development and easy deployment of small but high-value applications, including games and custom applications and services from the mobile phone manufacturer and network operator, all of which help to consolidate the position of Symbian OS in the market.

Java provides a powerful model for acquiring mass business market share and building a technical platform. One of the important, original underlying goals for Symbian OS was to enable Java as a true platform for developing Symbian OS applications, not simply as a lightweight platform for running relatively trivial generic applications.

Symbian (or Psion as it then was) was quick to recognize the significance of Java. The success of its own OPL language[2] and the emergence of a strong third-party developer community was an important element in the success of Psion's Series 3 products. The follow-on 32-bit system was conceived from the beginning as an open development platform for third-party application developers. At the same time, the complexity of its native C++ development environment and its unsuitability for some kinds of application development was recognized early, as was the need for a rapid development alternative. OPL support was present from the beginning, following which a Visual Basic porting project was started. However, attention moved to Java as a more powerful solution for third-party developers.

Sun's first release of Java appeared in early 1995 and the more stable JDK 1.0.2 release appeared in 1996. In early 1997, a few months before the first release of what was still known as EPOC32, Psion started its Java port based on JDK 1.1.2. 'Hello World' first ran in July 1997 and the first graphical application ran in October of that year. By the summer of 1998, graphical applications were running on an upgraded version of the port, based on JDK 1.1.4, and in August 1998 certification was granted by Sun. The first Java run-time system for EPOC32 was released in May 1999.[3] Demonstrations of Java applications running on early Symbian

[2] OPL lives on as a rapid development language for Symbian OS, having been released under an open source license by Symbian in 2003. It can be found on the web at ***www.allaboutopl.com/wiki/OPLWikiHome***. See also [Spence 2005].

[3] The full history of these early implementations can be found in earlier Symbian programming books, including [Tasker 2000] and [Allin *et al.* 2001].

OS smartphones caused quite a stir at the Symbian developer conference in June 1999.[4]

By that time, Psion had become Symbian and EPOC was evolving towards the first release of Symbian OS. However, porting Java had not been particularly easy. For one thing, Sun had at first insisted on a binary-only license for the VM. Since the VM assumed an ANSI C/POSIX platform, a basic C Standard Library implementation was needed to interface it to Symbian's native C++ APIs. Fundamental differences in the thread model between Java and Symbian OS posed further problems. And at that time, since the port was still based on 'full-sized' Java, graphics were AWT-based. On a small platform, it was simply not possible to extract acceptable performance from AWT.

Even before its first Java release for EPOC had shipped, Symbian had started work, early in 1999, on a port of PersonalJava, a newly defined, scaled-down Java specification targeting smaller devices. In July 2000, PersonalJava together with an implementation of the JavaPhone API was released as part of Symbian OS v6, appearing for the first time in a phone product later that year in the Nokia 9210 Communicator. JavaPhone opened key Symbian OS APIs to Java applications, giving basic access to the underlying address book, communications services and telephony APIs.

PersonalJava was a first attempt to define a Java environment suitable for constrained devices such as PDAs and mobile phones, and a forerunner of what eventually became Java ME. By the time of the first Symbian OS v7 release, the Java ME specification had been defined in terms of MIDP/CLDC. MIDP 1.0 was included for the first time in Symbian OS v7 and subsequently back-ported to earlier releases, appearing in the Nokia 7650, based on Symbian OS v6.1. However, the PersonalJava and JavaPhone combination was still offered as an option to licensees.

Symbian OS has tracked the evolution of the MIDP specification (indeed, as a member of the MIDP Expert Group, it has played an active role in shaping it) through subsequent releases of the operating system, supporting MIDP 2.0 for the first time in Symbian OS v7.0s, and extending its support in Symbian OS v8 and v9 with the addition of important optional packages, as well as improving the compatibility, interoperability and completeness of Java ME technology by providing support for the JTWI standard and increasing developer productivity by supporting on-device debugging and standard integration with Java IDEs through the Unified Emulator Interface (UEI).

MIDP 2.0 is a significant enhancement of the original MIDP specification. In particular, it supports 'push' applications, in other words, MIDP applications that are launched in response to 'external' events (i.e. events

[4] The Java team demonstrated a Rubik Cube application, running in color on a Psion netBook, to an enthusiastic audience.

originating outside the application's own process), for example, alarms, incoming messages or other network events.

At the same time, the CLDC specification that defines the execution environment has evolved (again with Symbian participating as a CLDC Expert Group member). In practical terms, the most significant change is the move in Symbian OS v7.0s from the KVM to CLDC-HI 1.0, with HotSpot VM technology, and in Symbian OS v8 to CLDC-HI 1.1. The current CLDC1.1 configuration is likely to be the final configuration from Symbian, although licensees may continue to evolve their own extensions and track future evolution of the MIDP profile.

12.5 Architecture

In the Symbian OS system model (see Figure 12.1), Java ME is shown as a self-contained block spanning the UI Framework and Application Services layers, which emphasizes the external view of Java ME as an application platform.

The system model idealizes the representation of Java ME. As conventionally represented (see Figure 12.2), MIDP/CLDC forms a software stack that provides the execution environment for MIDlets; the CLDC configuration consists of the VM itself and the basic Java languages libraries; the MIDP Profile defines the frameworks for application support expected from the device class and the various packages (both required and optional) providing the application APIs.

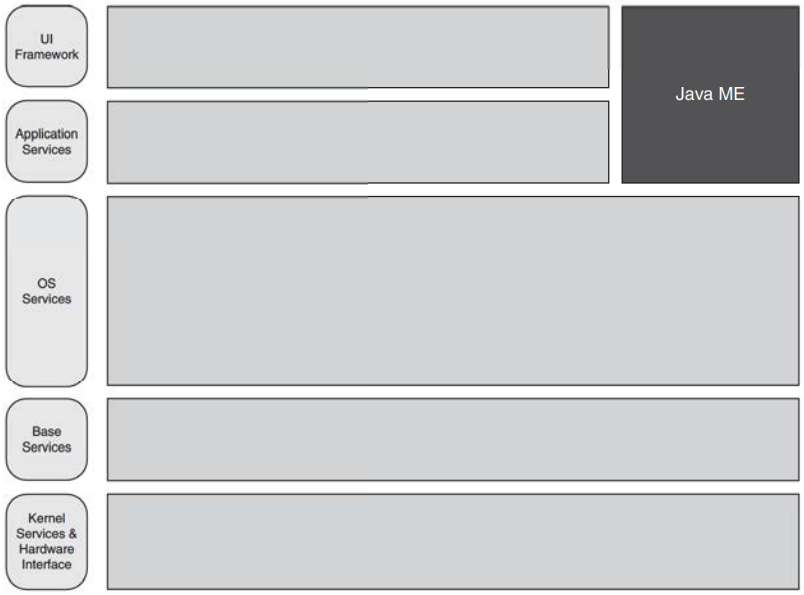


Figure 12.1 Java ME in the system model

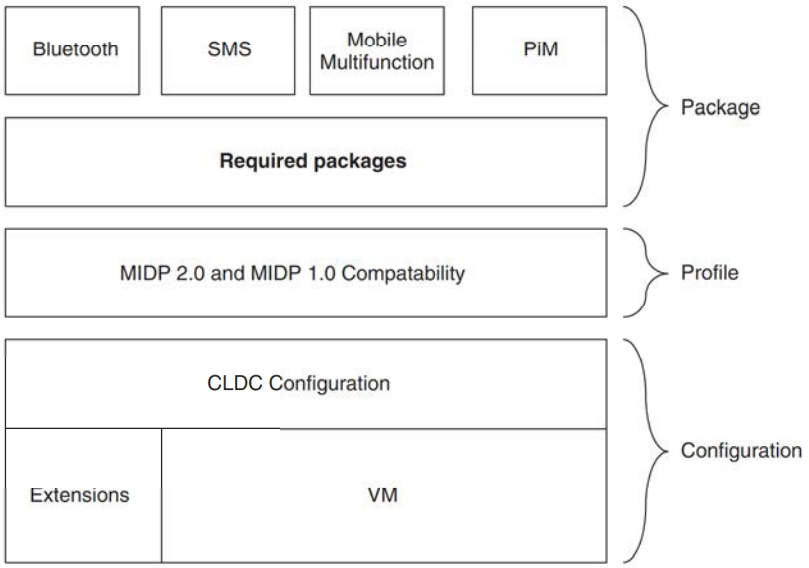


Figure 12.2 High-level Java ME architecture

From an internal architectural perspective, however, the Java ME implementation hooks deeply into the supporting layers of the operating system. Complex system interactions are required between the two, and indeed requirements originating from the successive Java implementations have had significant impact on the evolution of some fundamental features of the operating system, including the native Symbian OS application model, inter-process communication (IPC), and the thread and process models. In particular, the different threading models of Java and Symbian OS posed particularly thorny issues for the early Java implementations, which in turn influenced later design decisions.

At a high level, the architecture (see Figure 12.3) can be broken down as follows:

- the SystemAMS server, which is responsible for the lifecycle of MIDlets and VM processes, and static and dynamic resource management for Java applications
- the SystemAMS plug-ins for licensee customizability (i.e. customized security policy)
- the SystemAMS extension plug-ins that extend internal AMS frameworks
- the VM executable, which includes the VM, MIDP APIs and frameworks
- the VM plug-ins for licensee customizability (i.e. graphics customization)

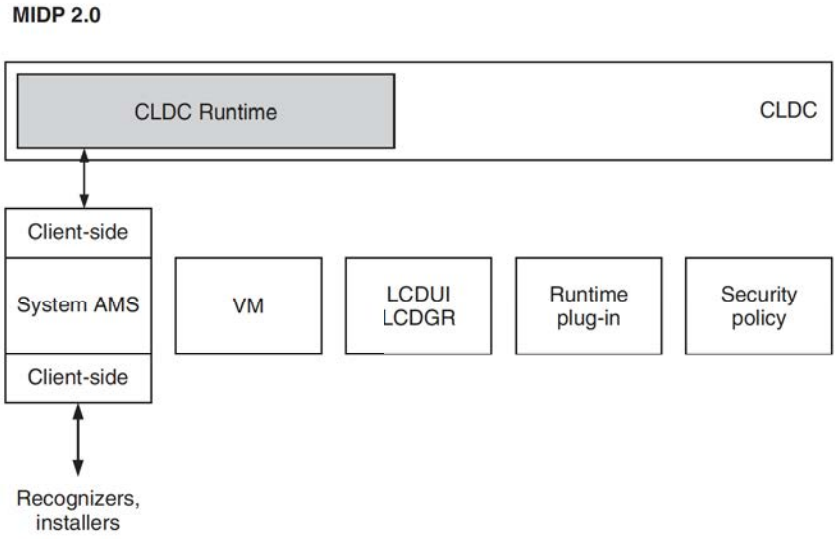


Figure 12.3 Internal architectural view of Java ME on Symbian OS

Java ME Applications Management Software

The System Application Management Software (SystemAMS)) operates as a managing agent between Symbian OS and the Java ME run-time (see Figure 12.4) for all stages of the MIDlet lifecycle from installation, launching and stopping MIDlets, controlling execution and launching VM processes as required. SystemAMS also manages static and dynamic push connections and alarms registered by MIDlets, which is the basis for the MIDP 2.0 'push' support.

SystemAMS is implemented as a server which is run from device boot time. From an application perspective, the MIDlet can initiate some state changes itself and notify the MIDP run time, which eventually notifies SystemAMS of those state changes by invoking the appropriate methods.

From a system perspective, SystemAMS provides client-side interfaces used by the VM and the Java installer to support MIDlet recognition, installation and launch; manages resources such as registered push connections and Symbian OS alarm notifications; and is responsible for managing the MIDP policy-security model.

The CLDC Configuration Layer

An important feature of Symbian's CLDC implementation is its support for various VMs. For example, in earlier versions of Symbian OS, the KVM was used and later replaced by the CLDC 1.0 VM which was eventually replaced by the CLDC 1.1 VM. As VMs may change, an abstraction layer is required between the VM and the various CLDC and MIDP

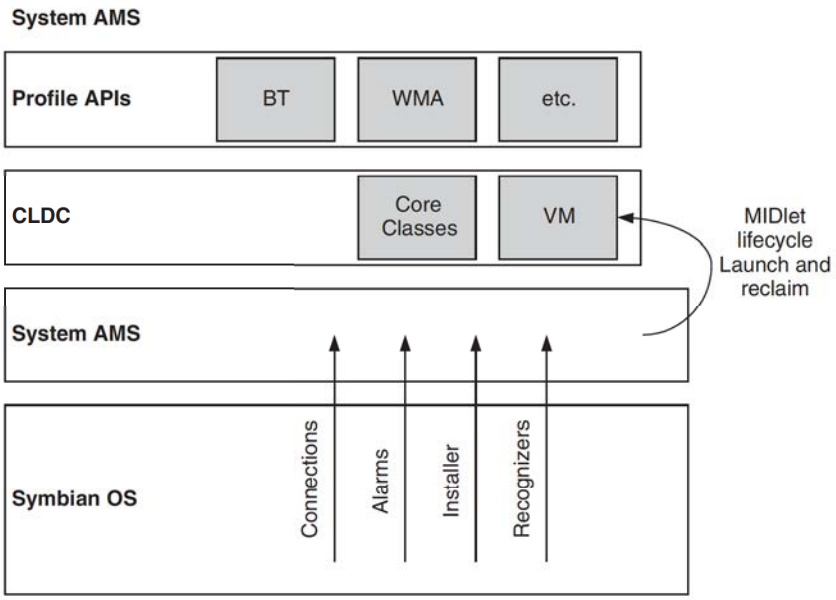


Figure 12.4 SystemAMS Architecture

libraries to avoid dependencies between them and any particular VM. The abstraction layer also interfaces Java event-handling with the native event-handling model.

CLDC-HI is designed as a high-performance JVM and run-time environment for resource-constrained small devices, in particular mobile phones and communicator-style devices, and is optimized for performance, footprint, and efficient resource management, with a specialized Just In Time (JIT) compiler for the ARM processor architecture. Sun claims an order of magnitude improvement over the performance of the older KVM, for example.

The MIDP Profile Layer

LCDUI

MIDP is specifically targeted at small, mobile devices. It includes the LCDUI specification, which is optimized for this device class. LCDUI defines both the UI event model and the standard GUI widgets available to MIDlets (lists, forms, textboxes, and so on). LCDUI therefore requires integration both with the native UI Framework and Application Architecture.

In order to make MIDlets (as far as possible) indistinguishable from native applications, LCDUI uses the native widget set as peers for the Java widgets. A MIDlet runs as a single native application owning its

own window group, listed in and launchable from the task list (if the user interface variant has a task list) and integrated with the save notifier framework, power events, and foreground/background notification. Input methods are also inherited from the native widget set so that all native functionality is available to MIDlets, for example, mechanisms such as T9, handwriting recognition and non-keyboard character set input (e.g. Chinese), which are all based on the Front End Processor (FEP) framework. This also harnesses the native locale support, for example for bi-directional text and capitalization.

The LCDUI implementation consists of a framework that implements the core user interface functionality and provides the high-level interface between Java ME LCDUI APIs and the concrete user interface platform implementation areas that are implemented in separate graphics plug-ins (which licensees customize to provide integration with the graphics system of their specific user interface platform).

GCF

The MIDP Generic Connection Framework (GCF) provides the generic mechanism for creating a connection from a URI and enables a wide variety of connections including networking connections such as HTTP and HTTPS, sockets and server sockets, secure sockets and datagrams, as well as support for 'push' connections and on-device mechanisms for local file and directory access (which is known as 'File GCF').

The MIDP GCF design maps the Java class interfaces to the underlying Symbian OS communications models and provides core communications functionality for MIDlets including:

- opening, closing, and disposing of connections
- opening Java streams for appropriate types
- a server connection pattern for types capable of receiving incoming connections.

Symbian's Java ME implementation enables all the relevant MIDP 2.0 GCF protocols and its framework is intended to be used by extensions that provide support for future protocols. In particular, it supports push connections, using the SystemAMS dynamic and static resource management for managing the registered connections.

Mobile Multimedia

Mobile Multimedia implements access to the multimedia support provided in the underlying Symbian OS, enabling MIDlets to play and record audio and video data from a wide range of inputs using a

range of possible mechanisms, including streaming. The design follows a framework-plug-in architecture:

- the framework provides the high-level interface to MIDP Multimedia functionality
- the reference DLL contains all dependencies on the underlying native multimedia services and can be customized.

PIM and RMS

PIM support is provided for accessing native Symbian OS contacts (i.e. phone book or address book) and agenda (i.e. calendar) entries, including Event and ToDo classes.

Record Management System (RMS) support, which enables MIDlets to store persistent data, is implemented over the native Symbian OS DBMS APIs, using the DBMS in client–server mode and thus enabling database-like functionality including transaction integrity and sharing between multiple clients for Java applications.

Security

SystemAMS and the MIDP run-time are responsible for supporting the MIDP security model, through static (at installation time) and dynamic (at run time) checking of permissions, which provides for trusted and untrusted MIDlets, and for protection based on security domains. Licensees implement specific security policies by customizing the MIDP security plug-ins.

12.6 Component Collections

The system model divides the Java ME block into a number of separate collections (see Figure 12.5), broadly layered to reflect the conventional layering of the Java ME software stack.

The foundation is provided by the core Java class implementations and the CLDC-HI VM, together with the low-level plug-ins that integrate the MIDP frameworks with Symbian OS. The MIDP profile and packages collections are layered over this foundation.

MIDP 2.0 Profile Collection

This collection (see Figure 12.6) implements the Java ME MIDP 2.0 Profile.

- The MIDP MIDlet component implements the MIDlet lifecycle, which defines how MIDlets are started, paused and destroyed and how they interact with the host environment.

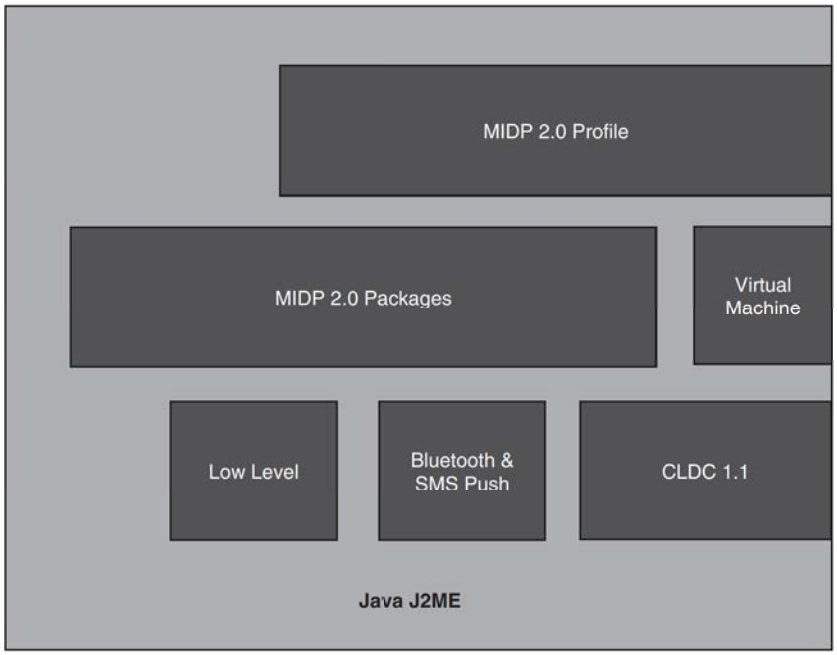


Figure 12.5 Java ME Block

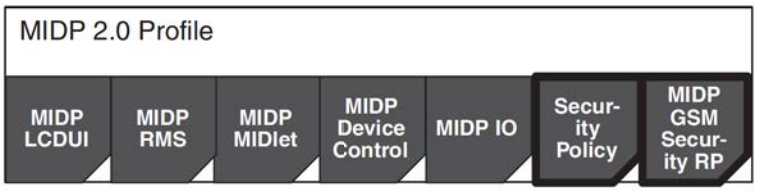


Figure 12.6 MIDP 2.0 Profile components

- The LCDUI component is specifically designed with small LCD screens in mind and provides compact, device-independent controls that can respond to user input ranging from keyboards to phone keys to touch screens. MIDP graphics APIs are implemented in terms of generic native controls which acquire platform-specific look and feel through the UI Application Framework implementation, which is customized by the UI variant.
- The RMS component provides MIDP persistence APIs. RMS is implemented internally over Symbian OS native DBMS, using the DBMS in client–server mode.
- The I/O component provides MIDP high-level input–output APIs, including networking support and HTTP connections.

Table 12.1 MIDP 2.0 Profile Components

Component Name	Development Name
MIDP MIDlet	MIDP2
MIDP LCDUI	JAVAX.MICROEDITION.LCDUI
MIDP RMS	JAVAX.MICROEDITION.RMS
MIDP IO	JAVAX.MICROEDITION.IO
MIDP Device Control	MIDP2
Security Policy Reference Plug-in	MIDP2SECURITY
MIDP GSM Security Recommended Policy	MIDP2SECURITYRP

- The Device Control component provides an interface for implementations of MIDP device control APIs, for example controlling device vibration.
- The Security Policy Reference Plug-in provides a reference implementation of Java security policy, implemented as a replaceable plug-in.
- The GSM Security Recommended Policy component adds specific protection domains to the security model (for example 'manufacturer', 'operator', 'third-party' and 'untrusted'). Licensees should customize and provide their own concrete implementation plug-in to be used by the framework.

MIDP 2.0 Packages Collection

The Java ME MIDP 2.0 packages components (see Figure 12.7) extend the MIDP 2.0 Profile implementation with additional APIs.

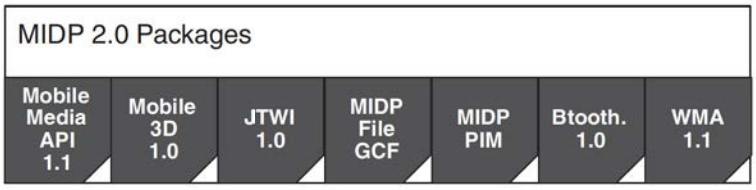


Figure 12.7 MIDP 2.0 Packages components

Table 12.2 MIDP 2.0 Packages Components

Component Name	Development Name
Mobile Media API 1.1	MMAPI11
Mobile 3D 1.0	M3GIO
JTWI 1.0	Java ME9.12
MIDP File GCF	GCF
MIDP PIM	MIDP2
Bluetooth 1.0	BLUETOOTH
WMA 1.1	WMA

- The Mobile Media API 1.1 component comprises Mobile Media APIs (JSR-135).
- The Mobile 3D 1.0 component comprises 3D-graphics APIs for scalable, small-footprint devices (JSR-184).
- The JTWI component implements the JTWI specification, which improves the compatibility, interoperability and completeness of Java ME technology implementations in mobile phones by reducing API fragmentation and raising the bar of functionality to specify a common set of APIs and standards such as MIDP 2.0 and including optional APIs (WMA 1.1 and MMAPI 1.1).
- The MIDP PIM and File GCF components support MIDP personal-information-management (PIM) and file-connection APIs (JSR-075), enabling reading and writing of event, contact, and to-do items, and file system access. File system access is implemented through the GCF communications framework, which generalizes framework support for HTTP, IP and socket-based connections.
- The Bluetooth component implements two optional MIDP2.0 Bluetooth 1.0 (JSR-082) packages: the core Bluetooth API and the Object Exchange (OBEX) API.
- The Wireless Messaging API (WMA) provides platform-independent access to wireless communication resources, enables send and receive of SMS messaging, including SMS push, on GSM, CDMA and other networks supporting asynchronous messaging protocols.

Both Bluetooth 1.0 and WMA 1.1 add 'push' capability to the support for MIDlets, allowing a MIDlet to respond either statically (at install time)



Figure 12.8 CLDC 1.1 components

or dynamically (programmatically) to an incoming WMA or Bluetooth connection (i.e. to a 'message'). Both implementations are integrated into the GCF communications framework.

CLDC 1.1 Collection

This component implements CLDC 1.1 Java class libraries (JSR-118). CLDC 1.1 specifies the core subset of the Java language required to support MIDlets. The language libraries define basic types and objects, including Byte, Integer, Object and Thread; the I/O libraries define the data-stream-based input–output APIs, as well as APIs for reading and writing bytes and basic Java types; the utilities library supplies basic utility classes, including Date and Time, and collection classes including Hashtable, Stack and Vector (see Figure 12.8).

Table 12.3 CLDC 1.1 Components

Component Name	Development Name
Java Lang	JAVA.LANG
Java IO	JAVA.IO
Java Utilities	JAVA.UTIL

Virtual Machine Collection

The CLDC-HI 1.1 component is the Sun CLDC HotSpot Implementation VM, which is part of the CLDC 1.1 specification (JSR-139). The HotSpot

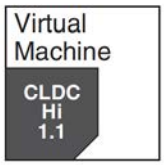


Figure 12.9 Virtual Machine components

Table 12.4 Virtual Machine Components

Component Name	Development Name
CLDC HI 1.1	CLDCHI

VM applies a variety of advanced performance-optimization techniques to deliver a highly competitive execution environment for Java applications. See Figure 12.9.

Low-Level Plug-ins Collection

These plug-ins allow customization of the CLDC run-time framework and bind LCDUI to the underlying graphics system. See Figure 12.10.

Table 12.5 Low-Level Plug-ins

Component Name	Development Name
Runtime Plug-in	MIDP2RUNTIME
LCDUI Plug-in	LCDUIB

- The Runtime plug-in component is the MIDP 2.0 run-time plug-in module. It can be customized by the licensee.
- The LCDUI plug-in component consists of low-level graphics APIs with direct screen access, implemented as a plug-in that is replaced with an alternative implementation.

Bluetooth and SMS Push Collection

These plug-ins bind the Bluetooth 1.0 and WMA 1.1 packages to the underlying system. See Figure 12.11.

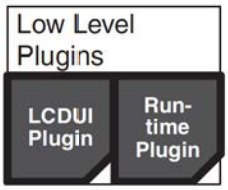


Figure 12.10 Low-Level Plug-ins

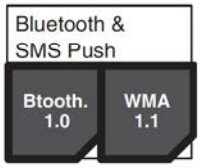


Figure 12.11 Bluetooth and SMS Push components

Table 12.6 Bluetooth and SMS Push Components

Component Name	Development Name
Bluetooth 1.0 Push Plug-in	BLUETOOTH
WMA 1.1. Push Plug-in	WMA

13

Notes on the Evolution of Symbian OS

13.1 The State of the Art

Symbian OS reached market for the first time towards the end of 2000, following on from the Psion EPOC32 releases. The last release of EPOC32 was Release 5; the first release of Symbian OS was Symbian OS v6.0.

The most recent release is Symbian OS v9, but Symbian OS v8 remains very much an active platform on which new products are still being developed and brought to market. Phones based on earlier releases still ship in their millions, even though Symbian's licensees have moved up to the latest releases for new products. Indeed until relatively recently, phones based on Symbian OS v6.1 continued to ship in quantity, particularly in Japan. Since then, Japanese licensees have led the way in adopting the new real-time kernel architecture, and have brought to market new 3G phones based on Symbian OS v8.1b and are likely to follow with phones based on Symbian OS v9.

In other markets, licensees are shipping new phones based both on Symbian OS v9 (platform security, new real-time kernel) and v8 (original kernel architecture). Symbian OS v7 (Sony Ericsson P910, Motorola A1000 and FOMA M1000) remains a volume-selling release and phones based on Symbian OS v6.1 (N-Gage QD) are still shipping.

At the time of writing, in late 2006, phones are shipping on all releases from Symbian OS v6.1 to Symbian OS v9.1, although new product pipelines from licensees are based on Symbian OS v8 and v9.

13.2 Summary of Symbian OS v6 Releases

Symbian OS v6 was the immediate result of a major and long-running project, working with Nokia as lead licensee, to re-engineer and re-architect Symbian OS from its EPOC32 baseline (ER5U). EPOC32 did support some phone and messaging functions, for example 'two-box'

telephony solutions in which an EPOC-based PDA could use a GSM mobile phone as a dialup modem, as well as driving it directly to send SMS messages and synchronize with the SIM phone book. However, EPOC remained substantially PDA-centric. Even more importantly, its Eikon GUI was not suitable for phones.

Among the most significant changes in Symbian OS v6, therefore, was the refactoring of Uikon to support multiple user interface implementations, so called 'variant UIs', and the more general re-architecting of phone-centric functionality to suit a true phone operating system. The Symbian OS v6 system architecture was based on a component-based release model and representation.

Symbian OS v6.0

Symbian OS v6.0 was the common platform for what were branded as the Crystal and Quartz reference designs, in keeping with Symbian's DFRD strategy.

One Crystal-based product, the Nokia 9210 Communicator, was brought to market. No Quartz devices reached market (although devices from Ericsson and Sanyo were demonstrated, including at 3GSM in Cannes).

The system architecture of Symbian OS v6.0 was based on a components representation inherited from the original Psion EPOC32 binary-component release model.

Symbian OS v6.1

Symbian OS v6.1 (announced in March 2001 at CeBIT) was the original platform for the Nokia S60 UI (which began life as the Pearl DFRD and launched as Series 60) The first Symbian OS v6.1/S60 phone (arguably, the first Symbian OS phone as opposed to PDA–phone hybrid) was the Nokia 7650. Other Symbian OS v6.1/S60 phones were brought to market by Panasonic, Sendo and Siemens.

Symbian OS v6.1 was very much an addition to the Symbian OS v6.0 baseline. No functionality was deprecated (although there were one or two significant reworkings); some functionality was added (around 150 new classes and other types). In almost all cases, changes were both binary and source compatible. Significant changes included:

- UI Framework and UI Toolkit changes and related changes to FEP and Text Formatting
- Application Services changes including new Contacts Model file format, new Chinese calendar and locale support, including Character Encoding Conversion updates, and improved VCard and VCalendar support

- OS Services changes including major Bluetooth revision (full Bluetooth 1.1 compliance), Infrared IrObex over Bluetooth, screen driver split out from BitGDI, 256-color-mode palette support added to Font & Bitmap Server, Free Type enhancements, Multimedia Server streaming added, new WAP Push and messaging functionality, telephony support for GSM/GPRS and SIM Toolkit, and Comms Database improvements.

13.3 Summary of Symbian OS v7 Releases

Symbian OS v7 was significant as the platform for the first UIQ phones. It also provided Symbian with its first real taste of the problems of fragmentation, with the divergence of Symbian OS v7.0 from Symbian OS v6.1 threatening the common platform model for UIQ and S60, subsequently corrected by the Symbian OS v7.0s release.

Symbian OS v7.0 was the platform for the Sony Ericsson P800 family; Symbian OS v7.0s was the platform for Nokia phones beginning with the 6600 and remained the mainstream platform for Nokia phones until the 6630 3G phone was released.

Symbian OS v7.0

Symbian OS v7.0, announced in February 2002 at 3GSM, was the platform for the UIQ UI (an evolution of the Quartz DFRD). The first Symbian OS v7 UIQ phone was the Sony Ericsson P800 (announced in Q1 2002 and released in Q4 2002).

Symbian OS v7.0 was, at a functional level, substantially backwards-compatible with Symbian OS v6.1, however there were numerous compatibility breaks, as well as significant new functionality added and significant restructuring of the UI Framework components to improve the separation between the framework support and the overlaying user interface. The TechView reference user interface components were also introduced (although TechView never became a complete reference user interface implementation).

Symbian OS v 7.0 also significantly reworked the source tree, introducing a subsystem-based release model and representation. Subsequently, Symbian OS v7.0 became the baseline for the architecture representation based on the system model, which has become Symbian's standard architectural representation for releases from Symbian OS v7.0 forwards.

Among the most significant changes from Symbian OS v6.1 were:

- Application support
  - The Time/World application refactored into Alarm Server and World Server

- New Help file format; Agenda and Contact format changes (for vCard and vCal)
- System agent updated to support two-box system.
- Improvements to text handling and text views (formatting) support for user interface
- Microsoft Word and Excel converter support

- UI and Graphics
  - Further refactoring of Uikon to support user interface separation and pluggable Look-And-Feel
  - New standard controls including animation in menus, menu-highlighting options, variable-height list items, customizable text wrapping, line breaks, automatic hyphenation (editable windows), improved error handling, a generic dialog server, flip support, and many other minor enhancements
  - Graphics changes for anti-aliasing, key click and long keypress support, direct screen access and 2D hardware-acceleration support, hardware bitmaps, font name aliases, polygon fill, bitmap scaling, fading
  - Front End Processor Base optimizations
  - New TechView test user interface
- Multimedia
  - Media Server updates to support audio and video streaming and graphics acceleration
  - Improved audio codec support
- Comms and Telephony
  - Telephony re-architecture introduced Multimode ETel (in place of the Basic, Advanced and GPRS extensions), enabling CDMA support and performance improvements
  - New 3rd-party telephony API, non-third party APIs restricted
  - New SIM Toolkit phone applications support
  - New reference and test TSY implementations
- Networking
  - Dual v4/v6 IP stack introduced, networking support for packet telephony (GPRS) and IPSec, including integration with new Multimode ETel
  - Socket Server changes relating to IPv6, internet sockets, and secure sockets

- Improved emulator Ethernet support

- Bluetooth and short link: Simplified HCI implementation to assist porting
- WAP and browsing
  - WAP stack withdrawn (reliance on licensee implementations), WAP messaging implementation refactored
  - HTTP Client API added
  - WSP Adaptation Layer and Protocol Handler APIs
  - Opera-specific web-browser support component added
- Messaging
  - Support introduced for multimedia messaging (MMS) including SMIL message content markup. SMS messaging re-architected including support for Multimode (non-GSM) phone messaging refactored from ETel
  - Smaller fixes in internet mail, fax client and scheduled send
- Cryptography: support for x.509 parsing and ASM.1 library added
- Connectivity: support for SyncML connectivity protocol added
- MIDP JAVA ME introduced with fully compliant support for MIDlets on Symbian OS; PersonalJava enhancements but JNI compatibility maintained
- System Libraries
  - ECom (also known as 'Magic') Plug-in Framework introduced implementing new plug-in architecture
  - StringPool API factored out of Uikon and re-engineered
  - Support for non ROM-based localization added
  - Support for Shift-JIS (Japanese) character set added
- Kernel, Base Porting, and Build Tools
  - Emulator target build system migrated to Metrowerks CodeWarrior from Microsoft Visual C++
  - Added XScale processor support
  - New kits delivery model
  - New Backup and Shutdown server, USB support, MMC card support, power management improvements, performance improvements (speed and ROM footprint).

Symbian OS v7.0s

Symbian OS v7.0s, announced in April 2003 at Symbian's developer event, repaired the fragmentation resulting from incompatibilities between Symbian OS v7.0 and v6.1 and the scale of the Symbian OS v7.0 changes, which threatened to create permanent divergence between S60-based product lines from Nokia, and its licensees, and UIQ-based products (for example, from Sony Ericsson and Motorola). The Symbian OS v7.0s system architecture was based on a subsystem release model and representation, retrospectively updated to the architecture representation based on the system model.

13.4 Summary of Symbian OS v8 Releases

Announced at 3GSM in February 2004 and reaching the market in phones such as the Nokia 6630, the Symbian OS v8 release was a significant increment on Symbian OS v7. In particular, it marked the first appearance of the real-time kernel.

In large part, the feature set is common to both Symbian OS v8.0 and Symbian OS v8.1, except that Symbian OS v8.1 offers the option of the new kernel.

The main new functionality in Symbian OS v8 includes the following (by no means an exhaustive list):

- CDMA telephony support
- Multimedia Framework replacing Media Server
- new connectivity, data synchronization and device management services architectures
- new WAP stack architecture and implementation
- OpenGL ES vector graphics support
- new implementation of Certificate and Key Management
- new App Installer architecture (preparing the way for Symbian OS v9 Platform Security)
- new content-access and content-handling frameworks, supporting policy-based content management, that is, DRM
- new JAVA ME JSRS
- USB-device class support
- MMS support, including parsing of SMIL markup, and support for OBEX over Bluetooth and Infrared

- Improved VCard and VCal conversion
- New XML parsing framework

Symbian OS v8.0

Originally Symbian OS v8.0 was envisaged as the release which would introduce the EKA2 kernel option for the first time. However, in the event, only one Symbian OS v8.0 release was made, based on the original EKA1 Symbian OS kernel.

Symbian OS v8.0 marked a substantial increment on Symbian OS v7, with new features spanning most layers of the system:

- Communications and telephony changes including the new communications framework based on the Comms Root Server and MBufs, first stage of CDMA telephony support, and Quality of Service (QoS) for GPRS
- Bluetooth and short-link changes introducing new USB class support providing control for USB devices and Bluetooth stack changes to support new Java ME JSRs
- New WAP short-stack WAP Messaging API, providing a limited functionality WAP stack and message API
- OpenGL ES Framework introduced, as well as multi-client access to screen, keyboard, and pointer or digitizer for GUI applications
- New Multimedia Framework replacing Media Server, new ECam camera API, Image Conversion Library and codec plug-ins, and low-level Media Device Framework providing low-level MIDI, video, speech recognition, and audio hardware-acceleration APIs
- New implementations of Certificate and Key Management and secure application installation
- Content-access and content-handling frameworks to support DRM content
- New connectivity, data synchronization and device management architectures
- Java ME new JSRs including Mobile 3D 1.0 (JSR-184), Bluetooth 1.0 (JSR082), Mobile Media API 1.1 (JSR135) and JTWI 1.0 (JSR185).
- New VCard and VCal conversion support and new character encodings
- New XML parsing framework, including XML Parsing Framework, WBXML Parser for WAP Binary XML, WBXML XML Parser Plugin
- Messaging support for OBEX over Bluetooth and MMS messaging

Symbian OS v8.1a

Symbian OS v8.1a is the Symbian OS v8.1 variant built on the original EKA1 kernel, but otherwise sharing the same features as Symbian OS v8.1b:

- ETel CDMA telephony extensions introduced support for CDMA networks, including CDMA SMS and WAP messaging support, and CDMA, Multimode, and SIM TSY reference plug-ins
- Graphics support for multiple simultaneous display, multiple display sizes and multiple display orientation
- SyncML device management
- Java ME upgrade to CLDC 1.1 from 1.0
- New Comms Database compatibility.

Symbian OS v8.1b

Symbian OS v8.1b is the real-time EKA2 kernel release of Symbian OS v8.1. This is the release, therefore, in which the EKA2 kernel becomes available for the first time.

In Japan, Symbian OS v8.1b has been the platform for a wave of 3G DoCoMo FOMA phones (from Fujitsu, Mitsubishi and Sharp), in particular with rich music and multimedia capabilities.

13.5 Summary of Symbian OS v9 Releases

Symbian OS v9 is the platform for the latest 'generation 3.x' UIQ and Nokia user interfaces, starting with UIQ 3 and S60 3rd Edition. It is also the release which introduces the new Platform Security architecture and completes the transition to the new real-time kernel architecture (first introduced as an option in Symbian OS v8.1; in Symbian OS v9, the original EKA1 kernel is retired).

Symbian OS v9 is the release with which Symbian has set its sights on the high-volume, mid-range market and the Symbian OS v9 releases to date take incremental steps to improve the fit of Symbian OS with that market, in particular in terms of performance and scalability (improving critical, basic performance and providing a cleaner architecture for porting to new hardware, support for single-core phone designs to reduce bill of materials (BOM) cost, improved peripherals support, and other porting and tools improvements to help time-to-market and reduce development cost). In keeping with these volume goals, it is also the platform on which Symbian has made its 'compatibility promise', the promise of API stability for all releases from Symbian OS v9 on.

Compared with Symbian OS v8, the headline changes are the retiring of the EKA1 kernel architecture, in favor of the new, real-time kernel, and the introduction of platform security, providing a trusted application model.

From a developer perspective, since the new kernel maintains user-side API compatibility, the kernel changes have little application-level impact. In contrast, platform-wide security has significant impact for all developers, introducing a security-capability model to protect system APIs and data caging for all application data.

The Symbian Signed signing and certification program grants capabilities (required to access protected APIs) to applications. Java ME MIDlets are also integrated into the security model. A free certification process is provided to freeware and shareware developers. Experience to date suggests that, for third-party developers, the more general certification requirements for robust and safe handling of extreme conditions (such as out-of-memory) are as much of an issue as the immediate requirements of the security model.

In other respects, Symbian OS v9 is very much an incremental update to Symbian OS v8. There is little radical architectural change, but a significant amount of re-engineering and improvement (in particular to improve performance, with boot time and RAM usage the key target areas, along with a set of user-oriented critical use-cases, for example addition and deletion of multiple contacts).

Symbian OS v9 also makes the transition from GCC to the ARM RVCT compiler, supporting the ARM ABI versions 1 and 2 and therefore promising tools interoperability for ABI-compliant tools (including compilers).

Future Symbian OS v9 releases are likely to continue the focus on performance, including high-performance graphics and support for continued user interface evolution, improved suitability for the mid-range, evolution of the build toolchain, and backwards compatibility, while introducing headline new technologies (likely candidates include location-based services).

Symbian OS v9.0

There is no productized Symbian OS v9.0 release, which instead served as a baseline release for the integration of Platform Security on top of the EKA2 kernel. All Symbian OS v9 'new features' (as opposed to platform changes) therefore appear in subsequent releases, from Symbian OS v9.1.

Symbian OS v9.1

The first phones based on Symbian OS v9.1 (which was announced in February 2005) shipped during the first half of 2006. They included the

Nokia N80 3G phone and Sony Ericsson P990, both with Wi-Fi and multimegapixel cameras (3 and 2 megapixels respectively).

For the enterprise market, device-provisioning enhancements and group-scheduling APIs are important additions.

Headline features include:

- Platform Security
- Real-time EKA2 kernel, with support for a range of ARM CPU architectures and memory models
- System Starter, a new policy-based mechanism for server startup
- New Device Management and Client Provisioning services, including a new Client Provisioning Framework
- Broadcast Tuner APIs
- Bluetooth Remote Control Framework
- Improved Timezone support
- Networking enhancements including RTP/RTCP support
- Java ME MIDP 2.0 (JSR 118) and CLDC 1.1 (JSR 139), plus new MIDlet security policy support
- Hardware reference platform incremented to TI OMAP H2.

Symbian OS v9.2

Symbian OS v9.2 continues performance improvements, supports several new locales and provides build-toolchain and porting improvements, while maintaining baseport compatibility, all of which can be seen as steps along the way to an improved mid-range offering.

New features include some important new APIs and technologies (the SIP Framework, for example). The major platform features, however, are those which it shares with Symbian OS v9.1: platform security and the EKA2 kernel architecture.

Compared with Symbian OS v9.1, the key changes are:

- New RTP/RTCP and SIP multimedia protocols, together with underlying networking support and communications architecture evolution to support high data rates
- New Hindi and Vietnamese and improved Japanese language and locale support
- Device management, messaging, email and multimedia enhancements

- Continued improvements in boot time, RAM usage, and range of specific performance use cases
- Hardware reference platform incremented to ARMv6.

Symbian OS v9.3

At the time of writing, the latest release (announced in July 2006) is Symbian OS v9.3. Compared with Symbian OS v9.2, the key changes are:

- additions to SIP protocols and continued evolution of the communications architecture
- support for Wi-Fi wireless networking.

Part 3

Design Case Studies

The case studies presented here provide an in-depth examination of significant turning points in the evolution of Symbian OS or of significant aspects of the wider context in which the operating system continues to evolve. They take an unashamedly historical approach, which I hope helps to capture the flavor of the times, as well as providing some insight into how real systems get made and what forces – often contingent and accidental – shape them.

Each study explores a different aspect of the evolution of Symbian OS: the adoption of object-oriented ideas in designing and creating the system, the choice of C++ as the implementation language and the consequences of that choice; the early decision to implement telephony support and what that meant for Symbian OS; the radical solution to the question of user interface customization and what led up to it and shaped it; the challenges of renewal and evolution, which all software systems face and which all system designers must overcome; and, finally, taking a small step back from the operating system itself to the wider context of its production, the tensions exposed by success and the scaling up from small-scale to large-scale software production and what it implies for the future of the system.

These chapters are exploratory and not always conclusive but, I hope, illuminating nonetheless.